**PL/SQL PROGRAMS**

**BASIC PROGRAMS**

PROGRAM 1: PL/SQL Program To Add Two Numbers

This program takes two inputs one for each variable and adds the result to a third variable and prints it.

Declare

Var1 integer;

Var2 integer;

Var3 integer;

Begin

Var1:=&var1;

Var2:=&var2;

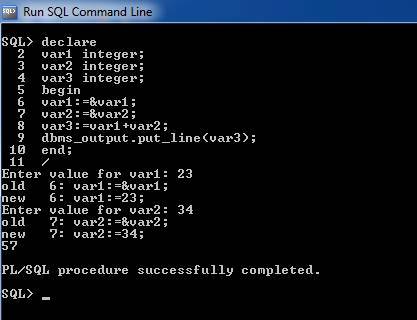
Var3:=var1+var2;

Dbms\_output.put\_line(var3);

End;

/

**Output**



Since, we need variables in this program code so we need to declare them in the declaration section. In the begin section, we need to initialize and take input from the user. All the logical and mathematical calculations for this program logic need to be established here.

**Var1:=&var1;**  
This statement assigns the value that the user enters for the variable. Similarly, we have taken input for next variable.

**Var3:=var1+var2**  
This statement is used to assign the calculated values of Var1 and Var2 into the third variable.

**Dbms\_output.put\_line()**  
The statement takes a parameter which can be printed onto the console screen.

**Note:** For this statement to work properly, it is sometimes necessary to use a particular statement as mentioned below:

When you start the SQL Command Prompt/Terminal, just type:  
  
**Set serveroutput on;**

This statement is used to activate the working of print statement on the console screen. This will help to get the proper output onto the screen/terminal as expected.

Finally, the End statement is used to terminate the code.

# PROGRAM 2: PL/SQL Program for Prime Number

A number is a prime number if it is divisible by 1 or itself. For example 2, 3, 5, 7, etc are prime numbers.

While numbers like 4, 6, 8, etc are not prime.

declare

    n number;

    i number;

    flag number;

begin

    i:=2;

    flag:=1;

    n:=&n;

    for i in 2..n/2

    loop

        if mod(n,i)=0

        then

            flag:=0;

            exit;

        end if;

    end loop;

    if flag=1

    then

        dbms\_output.put\_line('prime');

    else

        dbms\_output.put\_line('not prime');

    end if;

end;

/

**Output**

Enter value for n: 12  
old 9: n:=&n;  
new 9: n:=12;  
not prime

# pROGRAM 3: PL/SQL Program for Fibonacci Series

It is a series in which next number is the sum of previous two numbers.

declare

    first number:=0;

    second number:=1;

    third number;

    n number:=&n;

    i number;

begin

    dbms\_output.put\_line('Fibonacci series is:');

    dbms\_output.put\_line(first);

    dbms\_output.put\_line(second);

    for i in 2..n

    loop

        third:=first+second;

        first:=second;

        second:=third;

        dbms\_output.put\_line(third);

    end loop;

end;

/

**Output**

*Enter value for n: 6*  
*old 5: n number:=&n;*  
*new 5: n number:=6;*  
*Fibonacci series is:*  
*0*  
*1*  
*1*  
*2*  
*3*  
*5*  
*8*

# PROGRM 4: Pl/SQL Program for Palindrome Number

A number is called palindrome number if its reverse is equal to itself. For example 12321 is palindrome while 123 is not palindrome.

    declare

n number;

    m number;

    rev number:=0;

    r number;

begin

    n:=12321;

    m:=n;

    while n>0

    loop

        r:=mod(n,10);

        rev:=(rev\*10)+r;

        n:=trunc(n/10);

    end loop;

    if m=rev

    then

        dbms\_output.put\_line('number is palindrome');

    else

        dbms\_output.put\_line('number is not palindrome');

    end if;

end;

/

**Output**

*number is palindrome*

# PROGRAM 5:PL/SQL Program to Print Patterns

# **Pattern 1:**

\*

\*\*

\*\*\*

\*\*\*\*

\*\*\*\*\*

declare

    n number:=5;

    i number;

    j number;

begin

    for i in 1..n

    loop

        for j in 1..i

        loop

            dbms\_output.put('\*');

        end loop;

        dbms\_output.new\_line;

    end loop;

end;

/

# **Pattern 2:**

\*\*\*\*\*

\*\*\*\*

\*\*\*

\*\*

\*

declare

    n number:=5;

    i number;

    j number;

begin

    for i in reverse 1..n

    loop

        for j in 1..i

        loop

            dbms\_output.put('\*');

        end loop;

        dbms\_output.new\_line;

    end loop;

end;

/

# **Pattern 3:**

  \*

   \*\*

  \*\*\*

\*\*\*\*

\*\*\*\*\*

declare

    n number:=5;

    i number;

    j number;

    k number;

begin

    for i in 1..n

    loop

        for j in 1..n-i

        loop

            dbms\_output.put(' ');

        end loop;

        for k in 1..i

        loop

            dbms\_output.put('\*');

        end loop;

        dbms\_output.new\_line;

    end loop;

end;

/

# **Pattern 4:**

\*\*\*\*\*

\*\*\*\*

  \*\*\*

   \*\*

    \*

declare

    n number:=5;

    i number;

    j number;

    k number;

begin

    for i in reverse 1..n

    loop

        for j in 1..n-i

        loop

            dbms\_output.put(' ');

        end loop;

        for k in 1..i

        loop

            dbms\_output.put('\*');

        end loop;

        dbms\_output.new\_line;

    end loop;

end;

/

# **Pattern 5:**

\*

   \*\*\*

  \*\*\*\*\*

\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*

declare

    n number:=10;

    i number:=1;

    j number;

    k number;

begin

    while i<n

    loop

        j:=1;

        while j<n-i

        loop

            dbms\_output.put(' ');

            j:=j+2;

        end loop;

        for k in 1..i

        loop

            dbms\_output.put('\*');

        end loop;

        dbms\_output.new\_line;

        i:=i+2;

    end loop;

end;

/

# PL/SQL Procedure

**Introduction:** The PL/SQL stored procedure or simply a procedure is a PL/SQL block which performs one or more specific tasks. It is just like procedures in other programming languages.

The procedure contains a header and a body.

* **Header:** The header contains the name of the procedure and the parameters or variables passed to the procedure.
* **Body:** The body contains a declaration section, execution section and exception section similar to a general PL/SQL block.

## **How to pass parameters in procedure?**

When you want to create a procedure or function, you have to define parameters. There is three ways to pass parameters in procedure:

1. **IN parameters:**The IN parameter can be referenced by the procedure or function. The value of the parameter cannot be overwritten by the procedure or the function.
2. **OUT parameters:**The OUT parameter cannot be referenced by the procedure or function, but the value of the parameter can be overwritten by the procedure or function.
3. **INOUT parameters:**The INOUT parameter can be referenced by the procedure or function and the value of the parameter can be overwritten by the procedure or function.

## **Example 1:**

## **PL/SQL Create Procedure- Insert a record example using procedure**

To insert record in user table. So you need to create user table first.

**Step 1: Table creation:**

1. **create** **table** user(id number(10) **primary** **key**,**name** varchar2(100));

**Step 2: Procedure Code:**

1. **create** or replace **procedure** "INSERTUSER"
2. (id IN NUMBER,
3. **name** IN VARCHAR2)
4. **is**
5. **begin**
6. **insert** **into** user **values**(id,**name**);
7. **end**;
8. /

**Step3: the code to call above created procedure**.

1. **BEGIN**
2. insertuser(101,'Rahul');
3. dbms\_output.put\_line('record inserted successfully');
4. **END**;
5. /

## **Step4:** **PL/SQL Drop Procedure**

1. **DROP** **PROCEDURE** pro1;

**Example 2:**

Develop a procedure named adjust\_salary() in HR sample database provided by Oracle. update the salary information of employees in the employees table by using SQL UPDATE statement.

The following is the source code of the adjust\_salary() procedure :

**Step1:**

CREATE OR REPLACE PROCEDURE adjust\_salary(

    in\_employee\_id IN EMPLOYEES.EMPLOYEE\_ID%TYPE,

    in\_percent IN NUMBER

) IS

BEGIN

   -- update employee's salary

   UPDATE employees

   SET salary = salary + salary \* in\_percent / 100

   WHERE employee\_id = in\_employee\_id;

END;

**How it works.**

* The procedure has two parameters: IN\_EMPLOYEE\_ID and IN\_PERCENT.
* The procedure adjusts the salary of a particular employee specified the IN\_EMPLOYEE\_ID by a given percentage IN\_PERCENT.
* In the procedure body, we use the UPDATE statement to update the salary information.

## **Step 3: Calling PL/SQL Procedure**

A procedure can call other procedures. A procedure without parameters can be called directly by using EXEC statement or EXECUTE statement followed by the name of the procedure as follows:

EXEC procedure\_name();

EXEC procedure\_name;

**Step 4:** A procedure with parameters can be called by using EXECor EXECUTEstatement followed by procedure’s name and its parameters in the order corresponding to the parameters list of the procedure as shown below:

EXEC procedure\_name(param1,param2…paramN);

**Step 5:**Now, we can call adjust\_salary()procedure as the following statements: -- before adjustment

SELECT salary FROM employees WHERE employee\_id = 200;

-- call procedure

exec adjust\_salary(200,5);

-- after adjustment

SELECT salary FROM employees WHERE employee\_id = 200;

## Similarities between Procedure and Function

* Both can be called from other PL/SQL blocks.
* If the exception raised in the subprogram is not handled in the subprogram exception handling section, then it will propagate to the calling block.
* Both can have as many parameters as required.
* Both are treated as database objects in PL/SQL.

## Procedure Vs. Function: Key Differences

|  |  |
| --- | --- |
| **Procedure** | **Function** |
| * Used mainly to a execute certain process | * Used mainly to perform some calculation |
| * Cannot call in SELECT statement | * A Function that contains no DML statements can be called in SELECT statement |
| * Use OUT parameter to return the value | * Use RETURN to return the value |
| * It is not mandatory to return the value | * It is mandatory to return the value |
| * RETURN will simply exit the control from subprogram. | * RETURN will exit the control from subprogram and also returns the value |
| * Return datatype will not be specified at the time of creation | * Return datatype is mandatory at the time of creation |

**Functions**

**Syntax to create a function:**

1. **CREATE** [OR REPLACE] **FUNCTION** function\_name [parameters]
2. [(parameter\_name [IN | **OUT** | IN **OUT**] type [, ...])]
3. **RETURN** return\_datatype
4. {**IS** | **AS**}
5. **BEGIN**
6. < function\_body >
7. **END** [function\_name];

**Here:**

* **Function\_name:** specifies the name of the function.
* **[OR REPLACE]** option allows modifying an existing function.
* 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.

### **The function must contain a return statement.**

* RETURN clause specifies that 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.

## **Example 1: PL/SQL Function**

**Step 1:** Let's see a simple example to **create a function**.

1. **create** or replace **function** adder(n1 in number, n2 in number)
2. **return** number
3. **is**
4. n3 number(8);
5. **begin**
6. n3 :=n1+n2;
7. **return** n3;
8. **end**;
9. /

**Step 2:** **call the function**.

1. **DECLARE**
2. n3 number(2);
3. **BEGIN**
4. n3 := adder(11,22);
5. dbms\_output.put\_line('Addition is: ' || n3);
6. **END**;
7. /
8. **Output:**
9. Addition is: 33
10. Statement processed.
11. 0.05 seconds

## **Example 2 : PL/SQL Function**

## Example to demonstrate Declaring, Defining and Invoking a simple PL/SQL function which will compute and return the maximum of two values.

1. **DECLARE**
2. a number;
3. b number;
4. c number;
5. **FUNCTION** findMax(x IN number, y IN number)
6. **RETURN** number
7. **IS**
8. z number;
9. **BEGIN**
10. IF x > y **THEN**
11. z:= x;
12. **ELSE**
13. Z:= y;
14. **END** IF;
16. **RETURN** z;
17. **END**;
18. **BEGIN**
19. a:= 23;
20. b:= 45;
22. c := findMax(a, b);
23. dbms\_output.put\_line(' Maximum of (23,45): ' || c);
24. **END**;
25. /

**Output:**

Maximum of (23,45): 45

Statement processed.

0.02 seconds

## **Example 3: PL/SQL function using table**

Let's take a customer table. This example illustrates creating and calling a standalone function. This function will return the total number of CUSTOMERS in the customers table.

|  |  |  |  |
| --- | --- | --- | --- |
| **Customers** | | | |
| **Id** | **Name** | **Department** | **Salary** |
| 1 | alex | web developer | 35000 |
| 2 | ricky | program developer | 45000 |
| 3 | mohan | web designer | 35000 |
| 4 | dilshad | database manager | 44000 |

**Step 1: Create Function:**

1. **CREATE** OR REPLACE **FUNCTION** totalCustomers
2. **RETURN** number **IS**
3. total number(2) := 0;
4. **BEGIN**
5. **SELECT** count(\*) **into** total
6. **FROM** customers;
7. **RETURN** total;
8. **END**;
9. /
10. After the execution of above code, you will get the following result.
11. Function created.

**Step 2: Calling PL/SQL Function:**

While creating a function, you have to give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task. Once the function is called, the program control is transferred to the called function.

After the successful completion of the defined task, the call function returns program control back to the main program.

To call a function you have to pass the required parameters along with function name and if function returns a value then you can store returned value. Following program calls the function totalCustomers from an anonymous block:

1. **DECLARE**
2. c number(2);
3. **BEGIN**
4. c := totalCustomers();
5. dbms\_output.put\_line('Total no. of Customers: ' || c);
6. **END**;
7. /

After the execution of above code in SQL prompt, you will get the following result.

Total no. of Customers: 4

PL/SQL procedure successfully completed.

## **Example 4: PL/SQL Recursive Function**

## **Example to calculate the factorial of a number**

1. **DECLARE**
2. num number;
3. factorial number;
5. **FUNCTION** fact(x number)
6. **RETURN** number
7. **IS**
8. f number;
9. **BEGIN**
10. IF x=0 **THEN**
11. f := 1;
12. **ELSE**
13. f := x \* fact(x-1);
14. **END** IF;
15. **RETURN** f;
16. **END**;
18. **BEGIN**
19. num:= 6;
20. factorial := fact(num);
21. dbms\_output.put\_line(' Factorial '|| num || ' is ' || factorial);
22. **END**;
23. /

After the execution of above code at SQL prompt, it produces the following result.

Factorial 6 is 720

PL/SQL procedure successfully completed.

**PL/SQL Drop Function**

**Syntax for removing your created function:**

If you want to remove your created function from the database, you should use the following syntax.

1. **DROP** **FUNCTION** function\_name;

# PL/SQL Cursor

When an SQL statement is processed, Oracle creates a memory area known as context area. A cursor is a pointer to this context area. It contains all information needed for processing the statement. In PL/SQL, the context area is controlled by Cursor. A cursor contains information on a select statement and the rows of data accessed by it.

A cursor is used to referred to a program to fetch and process the rows returned by the SQL statement, one at a time. There are two types of cursors:

* Implicit Cursors
* Explicit Cursors

## **1) PL/SQL Implicit Cursors**

The implicit cursors are automatically generated by Oracle while an SQL statement is executed, if you don?t use an explicit cursor for the statement.

These are created by default to process the statements when DML statements like INSERT, UPDATE, DELETE etc. are executed.

Orcale provides some attributes known as Implicit cursor?s attributes to check the status of DML operations. Some of them are: %FOUND, %NOTFOUND, %ROWCOUNT and %ISOPEN.

**For example:**When you execute the SQL statements like INSERT, UPDATE, DELETE then the cursor attributes tell whether any rows are affected and how many have been affected. If you run a SELECT INTO statement in PL/SQL block, the implicit cursor attribute can be used to find out whether any row has been returned by the SELECT statement. It will return an error if there no data is selected.

The following table specifies the status of the cursor with each of its attribute.

|  |  |
| --- | --- |
| **Attribute** | **Description** |
| %FOUND | Its return value is TRUE if DML statements like INSERT, DELETE and UPDATE affect at least one row or more rows or a SELECT INTO statement returned one or more rows. Otherwise it returns FALSE. |
| %NOTFOUND | Its return value is TRUE if DML statements like INSERT, DELETE and UPDATE affect no row, or a SELECT INTO statement return no rows. Otherwise it returns FALSE. It is a just opposite of %FOUND. |
| %ISOPEN | It always returns FALSE for implicit cursors, because the SQL cursor is automatically closed after executing its associated SQL statements. |
| %ROWCOUNT | It returns the number of rows affected by DML statements like INSERT, DELETE, and UPDATE or returned by a SELECT INTO statement. |

## **PL/SQL Implicit Cursor Example**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **NAME** | **AGE** | **ADDRESS** | **SALARY** |
| 1 | Ramesh | 23 | Allahabad | 20000 |
| 2 | Suresh | 22 | Kanpur | 22000 |
| 3 | Mahesh | 24 | Ghaziabad | 24000 |
| 4 | Chandan | 25 | Noida | 26000 |
| 5 | Alex | 21 | Paris | 28000 |
| 6 | Sunita | 20 | Delhi | 30000 |

**Create customers table and have records:**

Let's execute the following program to update the table and increase salary of each customer by 5000. Here, SQL%ROWCOUNT attribute is used to determine the number of rows affected:

**Create procedure:**

1. **DECLARE**
2. total\_rows number(2);
3. **BEGIN**
4. **UPDATE**  customers
5. **SET** salary = salary + 5000;
6. IF sql%notfound **THEN**
7. dbms\_output.put\_line('no customers updated');
8. ELSIF sql%found **THEN**
9. total\_rows := sql%rowcount;
10. dbms\_output.put\_line( total\_rows || ' customers updated ');
11. **END** IF;
12. **END**;
13. /

Output:

6 customers updated

PL/SQL procedure successfully completed.

Now, if you check the records in customer table, you will find that the rows are updated.

1. **select** \* **from** customers;

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **NAME** | **AGE** | **ADDRESS** | **SALARY** |
| 1 | Ramesh | 23 | Allahabad | 25000 |
| 2 | Suresh | 22 | Kanpur | 27000 |
| 3 | Mahesh | 24 | Ghaziabad | 29000 |
| 4 | Chandan | 25 | Noida | 31000 |
| 5 | Alex | 21 | Paris | 33000 |
| 6 | Sunita | 20 | Delhi | 35000 |

## **2) PL/SQL Explicit Cursors**

The Explicit cursors are defined by the programmers to gain more control over the context area. These cursors 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.

Following is the syntax to create an explicit cursor:

## **Syntax of explicit cursor**

Following is the syntax to create an explicit cursor:

1. **CURSOR** cursor\_name **IS** select\_statement;;

## **Steps:**

You must follow these steps while working with an explicit cursor.

1. Declare the cursor to initialize in the memory.
2. Open the cursor to allocate memory.
3. Fetch the cursor to retrieve data.
4. Close the cursor to release allocated memory.

## **1) Declare the cursor:**

It defines the cursor with a name and the associated SELECT statement.

**Syntax for explicit cursor decleration**

1. **CURSOR** **name** **IS**
2. **SELECT** statement;

## **2) Open the cursor:**

It is used to allocate memory for the cursor and make it easy to fetch the rows returned by the SQL statements into it.

**Syntax for cursor open:**

1. **OPEN** cursor\_name;

## **3) Fetch the cursor:**

It is used to access one row at a time. You can fetch rows from the above-opened cursor as follows:

**Syntax for cursor fetch:**

1. **FETCH** cursor\_name **INTO** variable\_list;

## **4) Close the cursor:**

It is used to release the allocated memory. The following syntax is used to close the above-opened cursors.

**Syntax for cursor close:**

1. **Close** cursor\_name;

## **PL/SQL Explicit Cursor Example**

Explicit cursors are defined by programmers to gain more control over the context area. It is defined in the declaration section of the PL/SQL block. It is created on a SELECT statement which returns more than one row.

Let's take an example to demonstrate the use of explicit cursor. In this example, we are using the already created CUSTOMERS table.

**Create customers table and have records:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **NAME** | **AGE** | **ADDRESS** | **SALARY** |
| 1 | Ramesh | 23 | Allahabad | 20000 |
| 2 | Suresh | 22 | Kanpur | 22000 |
| 3 | Mahesh | 24 | Ghaziabad | 24000 |
| 4 | Chandan | 25 | Noida | 26000 |
| 5 | Alex | 21 | Paris | 28000 |
| 6 | Sunita | 20 | Delhi | 30000 |

**Create procedure:**

Execute the following program to retrieve the customer name and address.

1. **DECLARE**
2. c\_id customers.id%type;
3. c\_name customers.**name**%type;
4. c\_addr customers.address%type;
5. **CURSOR** c\_customers **is**
6. **SELECT** id, **name**, address **FROM** customers;
7. **BEGIN**
8. **OPEN** c\_customers;
9. LOOP
10. **FETCH** c\_customers **into** c\_id, c\_name, c\_addr;
11. EXIT **WHEN** c\_customers%notfound;
12. dbms\_output.put\_line(c\_id || ' ' || c\_name || ' ' || c\_addr);
13. **END** LOOP;
14. **CLOSE** c\_customers;
15. **END**;
16. /

Output:

1 Ramesh Allahabad

2 Suresh Kanpur

3 Mahesh Ghaziabad

4 Chandan Noida

5 Alex Paris

6 Sunita Delhi

PL/SQL procedure successfully completed.

# PL/SQL Exception Handling

## **What is Exception**

An error occurs during the program execution is called Exception in PL/SQL.

PL/SQL facilitates programmers to catch such conditions using exception block in the program and an appropriate action is taken against the error condition.

There are two type of exceptions:

* System-defined Exceptions
* User-defined Exceptions

## **PL/SQL Exception Handling**

**Syntax for exception handling:**

Following is a general syntax for exception handling:

1. **DECLARE**
2. <declarations **section**>
3. **BEGIN**
4. <executable command(s)>
5. EXCEPTION
6. <exception handling goes here >
7. **WHEN** exception1 **THEN**
8. exception1-handling-statements
9. **WHEN** exception2  **THEN**
10. exception2-handling-statements
11. **WHEN** exception3 **THEN**
12. exception3-handling-statements
13. ........
14. **WHEN** others **THEN**
15. exception3-handling-statements
16. **END**;

## **Example of exception handling**

Let's take a simple example to demonstrate the concept of exception handling. Here we are using the already created CUSTOMERS table.

SELECT\* FROM COUSTOMERS;

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **NAME** | **AGE** | **ADDRESS** | **SALARY** |
| 1 | Ramesh | 23 | Allahabad | 20000 |
| 2 | Suresh | 22 | Kanpur | 22000 |
| 3 | Mahesh | 24 | Ghaziabad | 24000 |
| 4 | Chandan | 25 | Noida | 26000 |
| 5 | Alex | 21 | Paris | 28000 |
| 6 | Sunita | 20 | Delhi | 30000 |

1. **DECLARE**
2. c\_id customers.id%type := 8;
3. c\_name  customers.**name**%type;
4. c\_addr customers.address%type;
5. **BEGIN**
6. **SELECT**  **name**, address **INTO**  c\_name, c\_addr
7. **FROM** customers
8. **WHERE** id = c\_id;
9. DBMS\_OUTPUT.PUT\_LINE ('Name: '||  c\_name);
10. DBMS\_OUTPUT.PUT\_LINE ('Address: ' || c\_addr);
11. EXCEPTION
12. **WHEN** no\_data\_found **THEN**
13. dbms\_output.put\_line('No such customer!');
14. **WHEN** others **THEN**
15. dbms\_output.put\_line('Error!');
16. **END**;
17. /

After the execution of above code at SQL Prompt, it produces the following result:

No such customer!

PL/SQL procedure successfully completed.

The above program should show the name and address of a customer as result whose ID is given. But there is no customer with ID value 8 in our database, so the program raises the run-time exception NO\_DATA\_FOUND, which is captured in EXCEPTION block.

#### **Note: You get the result "No such customer" because the customer\_id used in the above example is 8 and there is no cutomer having id value 8 in that table.**

If you use the id defined in the above table (i.e. 1 to 6), you will get a certain result. For a demo example: here, we are using the id 5.

1. **DECLARE**
2. c\_id customers.id%type := 5;
3. c\_name  customers.**name**%type;
4. c\_addr customers.address%type;
5. **BEGIN**
6. **SELECT**  **name**, address **INTO**  c\_name, c\_addr
7. **FROM** customers
8. **WHERE** id = c\_id;
9. DBMS\_OUTPUT.PUT\_LINE ('Name: '||  c\_name);
10. DBMS\_OUTPUT.PUT\_LINE ('Address: ' || c\_addr);
11. EXCEPTION
12. **WHEN** no\_data\_found **THEN**
13. dbms\_output.put\_line('No such customer!');
14. **WHEN** others **THEN**
15. dbms\_output.put\_line('Error!');
16. **END**;
17. /

After the execution of above code at SQL prompt, you will get the following result:

Name: alex

Address: paris

PL/SQL procedure successfully completed.

## **Raising Exceptions**

In the case of any internal database error, exceptions are raised by the database server automatically. But it can also be raised explicitly by programmer by using command RAISE.

**Syntax for raising an exception:**

1. **DECLARE**
2. exception\_name EXCEPTION;
3. **BEGIN**
4. IF condition **THEN**
5. RAISE exception\_name;
6. **END** IF;
7. EXCEPTION
8. **WHEN** exception\_name **THEN**
9. statement;
10. **END**;

## **PL/SQL User-defined Exceptions**

PL/SQL facilitates their users to define their own exceptions according to the need of the program. A user-defined exception can be raised explicitly, using either a RAISE statement or the procedure DBMS\_STANDARD.RAISE\_APPLICATION\_ERROR.

**Syntax for user define exceptions**

1. **DECLARE**
2. my-exception EXCEPTION;

**PL/SQL Trigger:**

## **Advantages of Triggers**

These are the following advantages of Triggers:

* Trigger generates some derived column values automatically
* Enforces referential integrity
* Event logging and storing information on table access
* Auditing
* Synchronous replication of tables
* Imposing security authorizations
* Preventing invalid transactions

## **Creating a trigger:**

**Syntax for creating trigger:**

1. **CREATE** [OR REPLACE ] **TRIGGER** trigger\_name
2. {BEFORE | **AFTER** | **INSTEAD** **OF** }
3. {**INSERT** [OR] | **UPDATE** [OR] | **DELETE**}
4. [**OF** col\_name]
5. **ON** table\_name
6. [REFERENCING OLD **AS** o NEW **AS** n]
7. [**FOR** EACH ROW]
8. **WHEN** (condition)
9. **DECLARE**
10. Declaration-statements
11. **BEGIN**
12. Executable-statements
13. EXCEPTION
14. Exception-handling-statements
15. **END**;

**Here,**

* CREATE [OR REPLACE] TRIGGER trigger\_name: It creates or replaces an existing trigger with the trigger\_name.
* {BEFORE | AFTER | INSTEAD OF} : This specifies when the trigger would be executed. The INSTEAD OF clause is used for creating trigger on a view.
* {INSERT [OR] | UPDATE [OR] | DELETE}: This specifies the DML operation.
* [OF col\_name]: This specifies the column name that would be updated.
* [ON table\_name]: This specifies the name of the table associated with the trigger.
* [REFERENCING OLD AS o NEW AS n]: This allows you to refer new and old values for various DML statements, like INSERT, UPDATE, and DELETE.
* [FOR EACH ROW]: This specifies a row level trigger, i.e., the trigger would be executed for each row being affected. Otherwise the trigger will execute just once when the SQL statement is executed, which is called a table level trigger.
* WHEN (condition): This provides a condition for rows for which the trigger would fire. This clause is valid only for row level triggers.

## **PL/SQL Trigger Example**

Let's take a simple example to demonstrate the trigger. In this example, we are using the following CUSTOMERS table:

**Create table and have records:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **NAME** | **AGE** | **ADDRESS** | **SALARY** |
| 1 | Ramesh | 23 | Allahabad | 20000 |
| 2 | Suresh | 22 | Kanpur | 22000 |
| 3 | Mahesh | 24 | Ghaziabad | 24000 |
| 4 | Chandan | 25 | Noida | 26000 |
| 5 | Alex | 21 | Paris | 28000 |
| 6 | Sunita | 20 | Delhi | 30000 |

**Create trigger:**

Let's take a program to create 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:

1. **CREATE** OR REPLACE **TRIGGER** display\_salary\_changes
2. BEFORE **DELETE** OR **INSERT** OR **UPDATE** **ON** customers
3. **FOR** EACH ROW
4. **WHEN** (NEW.ID > 0)
5. **DECLARE**
6. sal\_diff number;
7. **BEGIN**
8. sal\_diff := :NEW.salary  - :OLD.salary;
9. dbms\_output.put\_line('Old salary: ' || :OLD.salary);
10. dbms\_output.put\_line('New salary: ' || :NEW.salary);
11. dbms\_output.put\_line('Salary difference: ' || sal\_diff);
12. **END**;
13. /

After the execution of the above code at SQL Prompt, it produces the following result.

Trigger created.

**Check the salary difference by procedure:**

Use the following code to get the old salary, new salary and salary difference after the trigger created.

1. **DECLARE**
2. total\_rows number(2);
3. **BEGIN**
4. **UPDATE**  customers
5. **SET** salary = salary + 5000;
6. IF sql%notfound **THEN**
7. dbms\_output.put\_line('no customers updated');
8. ELSIF sql%found **THEN**
9. total\_rows := sql%rowcount;
10. dbms\_output.put\_line( total\_rows || ' customers updated ');
11. **END** IF;
12. **END**;
13. /

Output:

Old salary: 20000

New salary: 25000

Salary difference: 5000

Old salary: 22000

New salary: 27000

Salary difference: 5000

Old salary: 24000

New salary: 29000

Salary difference: 5000

Old salary: 26000

New salary: 31000

Salary difference: 5000

Old salary: 28000

New salary: 33000

Salary difference: 5000

Old salary: 30000

New salary: 35000

Salary difference: 5000

6 customers updated

**Note:** As many times you executed this code, the old and new both salary is incremented by 5000 and hence the salary difference is always 5000.

After the execution of above code again, you will get the following result.

Old salary: 25000

New salary: 30000

Salary difference: 5000

Old salary: 27000

New salary: 32000

Salary difference: 5000

Old salary: 29000

New salary: 34000

Salary difference: 5000

Old salary: 31000

New salary: 36000

Salary difference: 5000

Old salary: 33000

New salary: 38000

Salary difference: 5000

Old salary: 35000

New salary: 40000

Salary difference: 5000

6 customers updated

## **Important Points**

Following are the two very important point and should be noted carefully.

* OLD and NEW references are used for record level triggers these are not avialable for table level triggers.
* If you want to query the table in the same trigger, then you should use the AFTER keyword, because triggers can query the table or change it again only after the initial changes are applied and the table is back in a consistent state.

**Date of submission of the above programs**

|  |  |
| --- | --- |
| Date | Batch |
| 19th October 2018 | **B1 ( Friday)** |
| 19th October 2018 | **B2( Thursday)** |
| 16th october 2018 | **B3( Tuesday)** |

When the record is submitted, on the date specified: 10 marks would be awarded.

Things to be included in the record to get complete 10 marks:

1. Index column should contain the database name.
2. Date of experiment conducted and submitted should be duly filled.
3. Plsql programs should be executed and output should be included.

These are the pl sql programs for the final exams.

Along with sql queries will include 2 pl sql programs.

Final lab I want all of you to first write the er diagram, then syntax and finally query.

ANY DOUBTS PLEASE CONTACT, ONLY AFTER EXECUTING THE QUERIES.

…………………………………..ALL THE BEST………………………………………….