UNIT-4

Function:Creating and Dropping function

A function is same as a procedure except that it returns a value. Therefore, all the discussions of the previous chapter are true for functions too.

Creating a Function

A function is created using the **CREATE FUNCTION** statement. −

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

Where,

- *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.

Example

The following example illustrates how to create and call a standalone function. This function returns the total number of CUSTOMERS in the customers table.

We will use the CUSTOMERS table, which we had created

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;
/
```

When the above code is executed using the SQL prompt, it will produce the following result −

```
Function created.
```

Calling a Function

While creating a function, you 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. When a program calls a function, the program control is transferred to the called function.

A called function performs the defined task and when its return statement is executed or when the **last end statement** is reached, it returns the program control back to the main program.

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

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

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

```
Total no. of Customers: 6

PL/SQL procedure successfully completed.
```

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.
```

PL/SQL Recursive Functions

We have seen that a program or subprogram may call another subprogram. When a subprogram calls itself, it is referred to as a recursive call and the process is known as **recursion**.

To illustrate the concept, let us calculate the factorial of a number. 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
```

The following program calculates the factorial of a given number by calling itself recursively −

```
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;
/
```

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

```
Factorial 6 is 720

PL/SQL procedure successfully completed.
```

Purity levels in functions

PURITY LEVEL affects several things in oracle when code runs, but most notably it affects optimization.

When a piece of sql runs, oracle can optimize it many different ways. Some of the more advanced optimization techniques are not based on which index to use, but rather on how to rewrite a query to a sematically equivelant form that will run faster yet will still yeild a correct result. To this end there has been a great deal of research and thinking done by database vendors and oracle in particular to invent intelligent and creative ways to rewrite a query.

But as you can imagine, like any other theory, there are rules and requirements that must be adhered to in order to make certain that any rewrite created is fully correct. So here is the kicker... Many rewrite optimizations are valid only if certain facts are true about a query. PURITY LEVEL is one way to describe such facts. If a query satisfies only a low level of purity then some rewrite techniques can be used but not others. But if a query is at the highest level of purity, then all rewrite techniques can be used on the query without fear of creating a version of the query that is not semantically equivelant or which generates an incorrect answer.

So PURITY LEVEL affects query rewrite. The more pure the query, the more rewrite techniques that can be employed to rewrite it.

But what is PURITY LEVEL? One way to think about PURITY LEVEL is as a description of how a piece of code interacts with its environment. Here is a list of PURITY LEVEL names and what they mean:

Code: [Select all] [Show/ hide]

```
RNDS  --  read no database state
RNPS  --  read no package state
WNDS  --  write no database state
WNPS  --  write no package state
```

RNDS means that the covered piece of code does not select data from the database.

RNPS means that the covered piece of code does not read package global variables.

WNDS means that the covered piece of code does not insert/update/delete or otherwise change data in the database.

WNPS means that the covered piece of code does not change the values of global package variables.

These concepts apply to 3gl code like PLSQL and JAVA. If a PLSQL function for example, changes package global variables, then it cannot claim to support WNPS PURITY LEVEL. If a SELECT statement calls this package function then the calling SQL statement also cannot claim to support WNPS PURITY LEVEL because it must accept the limits of the PLSQL code it is calling. Thus query rewrite techniques that require a WNPS PURITY LEVEL cannot be used on this piece of SQL code.

Normally oracle can look at a piece of code and know what the PURITY LEVEL is so you do not have to tell oracle. But sometimes it can't and so you have to supply a RESTRICT_REFERENCES pragma to tell oracle what the PURITY LEVEL is. You do not have to worry about being wrong because if you are, then at runtime Oracle will figure that out and generate an error.

Either way, at compile time or at runtime, oracle will tell you if you need to specify a pragma for your code to be executable, or if what you specified is invalid. But again, most of the time you do not need to do anything. Indeed I have not used the RESTRICT_REFERNCES pragma since version 9i of the database.

If you want to know more detail, there is plenty of info available with a simple GOOGLE. But to answer your initial question directly:

in the early days (oracle8i) we had to specify the PURITY LEVEL of plsql code a lot because Oracle was not very good at figuring it out for itself. But with currently supported releases 9i, 10g, 11g, oracle is way better at it so we almost never specify the PURITY LEVEL of a plsql component. Oracle will tell you if you need to do so either at compile time or runtime.

Triggers

What is Trigger in PL/SQL?

TRIGGERS are stored programs that are fired by Oracle engine automatically when DML Statements like insert, update, delete are executed on the table or some events occur. The code to be excecuted in case of a trigger can be defined as per the requirement. You can choose the event upon which the trigger needs to be fired and the timing of the execution. The purpose of trigger is to maintain the integrity of information on the database.

Benefits of Triggers

Following are the 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

Types of Triggers in Oracle

Triggers can be classified based on the following parameters.

- Classification based on the **timing**
  - BEFORE Trigger: It fires before the specified event has occurred.
  - AFTER Trigger: It fires after the specified event has occurred.
  - INSTEAD OF Trigger: A special type. You will learn more about the further topics. (only for DML)
- Classification based on the **level**
  - STATEMENT level Trigger: It fires one time for the specified event statement.
  - ROW level Trigger: It fires for each record that got affected in the specified event. (only for DML)
- Classification based on the **Event**
  - DML Trigger: It fires when the DML event is specified (INSERT/UPDATE/DELETE)
  - DDL Trigger: It fires when the DDL event is specified (CREATE/ALTER)
  - DATABASE Trigger: It fires when the database event is specified (LOGON/LOGOFF/STARTUP/SHUTDOWN)

How to Create Trigger

Below is the syntax for creating a trigger.

Syntax:

```
CREATE [ OR REPLACE ] TRIGGER <trigger_name>

[ BEFORE | AFTER | INSTEAD OF ]

[ INSERT | UPDATE | DELETE.....]

ON <name of underlying object>

[ FOR EACH ROW ]

[ WHEN <condition for trigger to get execute> ]

DECLARE
    <Declaration part>
BEGIN
    <Execution part>
EXCEPTION
    <Exception handling part>
END;
```

Trigger Timing

Event

Row Level

Conditional Clause

Syntax Explanation:

- The above syntax shows the different optional statements that are present in trigger creation.
- BEFORE/ AFTER will specify the event timings.
- INSERT/UPDATE/LOGON/CREATE/etc. will specify the event for which the trigger needs to be fired.
- ON clause will specify on which object the above-mentioned event is valid. For example, this will be the table name on which the DML event may occur in the case of DML Trigger.
- Command "FOR EACH ROW" will specify the ROW level trigger.
- WHEN clause will specify the additional condition in which the trigger needs to fire.
- The declaration part, execution part, exception handling part is same as that of the other PL/SQL blocks. Declaration part and exception handling part are optional.

:NEW and :OLD Clause

In a row level trigger, the trigger fires for each related row. And sometimes it is required to know the value before and after the DML statement.

Oracle has provided two clauses in the RECORD-level trigger to hold these values. We can use these clauses to refer to the old and new values inside the trigger body.

- :NEW – It holds a new value for the columns of the base table/view during the trigger execution
- :OLD – It holds old value of the columns of the base table/view during the trigger execution

This clause should be used based on the DML event. Below table will specify which clause is valid for which DML statement (INSERT/UPDATE/DELETE).

	INSERT	**UPDATE**	**DELETE**
:NEW	VALID	VALID	INVALID. There is no new value in delete case.
:OLD	INVALID. There is no old value in insert case	VALID	VALID

For Example: The price of a product changes constantly. It is important to maintain the history of the prices of the products.

We can create a trigger to update the 'product_price_history' table when the price of the product is updated in the 'product' table.

1) Create the 'product' table and 'product_price_history' table

```
CREATE TABLE product_price_history
(product_id number(5),
product_name varchar2(32),
supplier_name varchar2(32),
unit_price number(7,2) );



CREATE TABLE product
(product_id number(5),
product_name varchar2(32),
supplier_name varchar2(32),
unit_price number(7,2) );
```

2) **Create the price_history_trigger and execute it**.

```
CREATE or REPLACE TRIGGER price_history_trigger
BEFORE UPDATE OF unit_price
ON product
FOR EACH ROW
BEGIN
INSERT INTO product_price_history
VALUES
(:old.product_id,
 :old.product_name,
 :old.supplier_name,
```

```
 :old.unit_price);
END;
/
```

3) **Lets update the price of a product**.

```
UPDATE PRODUCT SET unit_price = 800 WHERE product_id = 100
```

Once the above update query is executed, the trigger fires and updates the 'product_price_history' table.

4)If you ROLLBACK the transaction before committing to the database, the data inserted to the table is also rolled back.

Types of PL/SQL Triggers

There are two types of triggers based on the which level it is triggered.
1) Row level trigger - An event is triggered for each row upated, inserted or deleted.
2) Statement level trigger - An event is triggered for each sql statement executed.

PL/SQL Trigger Execution Hierarchy

The following hierarchy is followed when a trigger is fired.
1) BEFORE statement trigger fires first.
2) Next BEFORE row level trigger fires, once for each row affected.
3) Then AFTER row level trigger fires once for each affected row. This events will alternates between BEFORE and AFTER row level triggers.
4) Finally the AFTER statement level trigger fires.

For Example: Let's create a table 'product_check' which we can use to store messages when triggers are fired.

```
CREATE TABLE product
(Message varchar2(50),
 Current_Date number(32)
);
```

Let's create a BEFORE and AFTER statement and row level triggers for the product table.

1) BEFORE UPDATE, Statement Level: This trigger will insert a record into the table 'product_check' before a sql update statement is executed, at the statement level.

```
CREATE or REPLACE TRIGGER Before_Update_Stat_product
BEFORE
UPDATE ON product
Begin
INSERT INTO product_check
Values('Before update, statement level',sysdate);
END;
/
```

2) BEFORE UPDATE, Row Level: This trigger will insert a record into the table 'product_check' before each row is updated.

```
CREATE or REPLACE TRIGGER Before_Upddate_Row_product
BEFORE
UPDATE ON product
FOR EACH ROW
BEGIN
INSERT INTO product_check
Values('Before update row level',sysdate);
END;
/
```

3) AFTER UPDATE, Statement Level: This trigger will insert a record into the table 'product_check' after a sql update statement is executed, at the statement level.

```
CREATE or REPLACE TRIGGER After_Update_Stat_product
AFTER
UPDATE ON product
BEGIN
INSERT INTO product_check
Values('After update, statement level', sysdate);
End;
/
```

4) AFTER UPDATE, Row Level: This trigger will insert a record into the table 'product_check' after each row is updated.

```
CREATE or REPLACE TRIGGER After_Update_Row_product
AFTER
insert On product
FOR EACH ROW
BEGIN
INSERT INTO product_check
Values('After update, Row level',sysdate);
END;
/
```

Now lets execute a update statement on table product.

```
UPDATE PRODUCT SET unit_price = 800
WHERE product_id in (100,101);
```

Lets check the data in 'product_check' table to see the order in which the trigger is fired.

```
SELECT * FROM product_check;
```

Output:

```
Mesage                              Current_Date

-----------------------------------------------------------

Before update, statement level      26-Nov-2008
Before update, row level            26-Nov-2008
After update, Row level             26-Nov-2008
Before update, row level            26-Nov-2008
After update, Row level             26-Nov-2008
After update, statement level       26-Nov-2008
```

The above result shows 'before update' and 'after update' row level events have occured twice, since two records were updated. But 'before update' and 'after update' statement level events are fired only once per sql statement.

The above rules apply similarly for INSERT and DELETE statements.

How To know Information about Triggers.

We can use the data dictionary view 'USER_TRIGGERS' to obtain information about any trigger.

The below statement shows the structure of the view 'USER_TRIGGERS'

DESC USER_TRIGGERS;

```
NAME                    Type

-------------------------------------------------------

TRIGGER_NAME            VARCHAR2(30)
TRIGGER_TYPE            VARCHAR2(16)
TRIGGER_EVENT           VARCHAR2(75)
TABLE_OWNER             VARCHAR2(30)
BASE_OBJECT_TYPE        VARCHAR2(16)
TABLE_NAME              VARCHAR2(30)
COLUMN_NAME             VARCHAR2(4000)
REFERENCING_NAMES       VARCHAR2(128)
WHEN_CLAUSE             VARCHAR2(4000)
STATUS                  VARCHAR2(8)
DESCRIPTION             VARCHAR2(4000)
ACTION_TYPE             VARCHAR2(11)
TRIGGER_BODY            LONG
```

This view stores information about header and body of the trigger.

*SELECT * FROM user_triggers WHERE trigger_name = 'Before_Update_Stat_product';*

The above sql query provides the header and body of the trigger 'Before_Update_Stat_product'.

You can drop a trigger using the following command.

DROP TRIGGER trigger_name;

INSTEAD OF Trigger

"INSTEAD OF trigger" is the special type of trigger. It is used only in DML triggers. It is used when any DML event is going to occur on the complex view.

Consider an example in which a view is made from 3 base tables. When any DML event is issued over this view, that will become invalid because the data is taken from 3 different tables. So in this INSTEAD OF trigger is used. The INSTEAD OF trigger is used to modify the base tables directly instead of modifying the view for the given event.

Example 1: In this example, we are going to create a complex view from two base table.

- Table_1 is emp table and
- Table_2 is department table.

Then we are going to see how the INSTEAD OF trigger is used to issue UPDATE the location detail statement on this complex view. We are also going to see how the :NEW and :OLD is useful in triggers.

- Step 1: Creating table 'emp' and 'dept' with appropriate columns
- Step 2: Populating the table with sample values
- Step 3: Creating view for the above created table
- Step 4: Update of view before the instead-of trigger
- Step 5: Creation of the instead-of trigger
- Step 6: Update of view after instead-of trigger

```
1.  CREATE TABLE emp (
2.  emp_no NUMBER,
3.  emp_name VARCHAR2(50),
4.  salary NUMBER,
5.  manager VARCHAR2(50),
6.  dept_no NUMBER);
7.  /
```

Output:

Table created

```
8.  CREATE TABLE dept(
9.  Dept_no NUMBER,
10. Dept_name VARCHAR2(50),
11. LOCATION VARCHAR2(50));
12. /
```

Output:

Table created

```
CREATE TABLE emp(

emp_no NUMBER,

emp_name VARCHAR2(50),

salary NUMBER,

manager VARCHAR2(50),

dept_no NUMBER);

/



CREATE TABLE dept(

Dept_no NUMBER,

Dept_name VARCHAR2(50),

LOCATION VARCHAR2(50));

/
```

Code Explanation

- **Code line 1-7**: Table 'emp' creation.

- **Code line 8-12**: Table 'dept' creation.
  - **Output**
  - Table Created
  - **Step 2)** Now since we have created the table, we will populate this table with sample values and Creation of Views for the above tables.

```
13. BEGIN
14. INSERT INTO DEPT VALUES (10, 'HR', 'USA');
15. INSERT INTO DEPT VALUES (20, 'SALES', 'UK');
16. INSERT INTO DEPT VALUES (30, 'FINANCIAL', 'JAPAN');
17. COMMIT;
18. END;
19. /
```

Output:

PL/SQL procedure successfully completed

```
20. BEGIN
21. INSERT INTO EMP VALUES (1000, 'XXX', 15000, 'AAA',30);
22. INSERT INTO EMP VALUES (1001, 'YYY', 18000, 'AAA',20);
23. INSERT INTO EMP VALUES (1002, 'ZZZ',  20000, 'AAA',10);
24. COMMIT;
25. END;
26. /
```

Output:

PL/SQL procedure successfully completed

Code Explanation

- **Code line 13-19**: Inserting data into 'dept' table.
- **Code line 20-26:** Inserting data into 'emp' table.

Output

PL/SQL procedure completed

Step 3) Creating a view for the above created table.

```
27. CREATE VIEW guru99_emp_view(
28. Employee_name, dept_name, location) AS
29. SELECT emp.emp_name, dept.dept_name, dept.location
30. FROM emp, dept
31. WHERE emp.dept_no=dept.dept_no;
32. /
```

Output:

View created

```
33. SELECT * FROM guru99_emp_view;
```

Output:

EMPLOYEE_NAME	DEPT_NAME	LOCATION
ZZZ	HR	USA
YYY	SALES	UK
XXX	FINANCIAL	JAPAN


```
CREATE VIEW guru99_emp_view(

Employee_name:dept_name,location) AS

SELECT emp.emp_name,dept.dept_name,dept.location

FROM emp,dept

WHERE emp.dept_no=dept.dept_no;

/

SELECT * FROM guru99_emp_view;
```

Code Explanation

- **Code line 27-32:** Creation of 'guru99_emp_view' view.
- **Code line 33:** Querying guru99_emp_view.

Output

View created

EMPLOYEE_NAME	DEPT_NAME	LOCATION
ZZZ	HR	USA
YYY	SALES	UK
XXX	FINANCIAL	JAPAN

Step 4) Update of view before instead-of trigger.

```
34. BEGIN
35. UPDATE guru99_emp_view SET location='FRANCE' WHERE employee_name='XXX';
36. COMMIT;
37. END;
38. /
```

Output:

```
ORA-01779: cannot modify a column which maps to a non key-preserved table
ORA-06512: at line 2
```

Code Explanation

- **Code line 34-38:** Update the location of "XXX" to 'FRANCE'. It raised the exception because the DML statements are not allowed in the complex view.

Output

ORA-01779: cannot modify a column which maps to a non key-preserved table

ORA-06512: at line 2

Step 5) To avoid the error encounter during updating view in the previous step, in this step we are going to use "instead of trigger."

```
39. CREATE TRIGGER guru99_view_modify_trg
40. INSTEAD OF UPDATE
41. ON guru99_emp_view
42. FOR EACH ROW
43. BEGIN
44. UPDATE dept
45. SET location=:new.location
46. WHERE dept_name=:old.dept_name;
47. END;
48. /
```

Output:

```
Trigger created
```

```
CREATE TRIGGER guru99_view_modify_trg

INSTEAD OF UPDATE

ON guru99_emp_view
```

```
FOR EACH ROW

BEGIN

UPDATE dept

SET location=:new.location

WHERE dept_name=:old.dept_name;

END;

/
```

Code Explanation

- **Code line 39:** Creation of INSTEAD OF trigger for 'UPDATE' event on the 'guru99_emp_view' view at the ROW level. It contains the update statement to update the location in the base table 'dept'.
- **Code line 44:** Update statement uses ':NEW' and ': OLD' to find the value of columns before and after the update.

Output

Trigger Created

Step 6) Update of view after instead-of trigger. Now the error will not come as the "instead of trigger" will handle the update operation of this complex view. And when the code has executed the location of employee XXX will be updated to "France" from "Japan."

```
49. BEGIN
50. UPDATE guru99_emp_view SET location='FRANCE' WHERE employee_name='XXX';
51. COMMIT;
52. END;
53. /
Output:
    PL/SQL procedure successfully completed

54. SELECT * FROM guru99_emp_view;
Output:
    EMPLOYEE_NAME      DEPT_NAME LOCATION
    ZZZ                HR        USA
    YYY                SALES     UK
    XXX                FINANCIAL FRANCE
```

```
BEGIN
```

```
UPDATE guru99_emp_view SET location='FRANCE' WHERE employee_name='XXX';

COMMIT;

END;

/

SELECT * FROM guru99_emp_view;
```

Code Explanation:

- **Code line 49-53:** Update of the location of "XXX" to 'FRANCE'. It is successful because the 'INSTEAD OF' trigger has stopped the actual update statement on view and performed the base table update.
- **Code line 55:** Verifying the updated record.

Output:

PL/SQL procedure successfully completed

EMPLOYEE_NAME	DEPT_NAME	LOCATION
ZZZ	HR	USA
YYY	SALES	UK
XXX	FINANCIAL	FRANCE

Conditional Predicates for Detecting Triggering DML Statement

The triggering event of a DML trigger can be composed of multiple triggering statements. When one of them fires the trigger, the trigger can determine which one by using these **conditional predicates**:

Conditional Predicate	TRUE if and only if:
INSERTING	An INSERT statement fired the trigger.
UPDATING	An UPDATE statement fired the trigger.
UPDATING ('*column*')	An UPDATE statement that affected the specified column fired the trigger.
DELETING	A DELETE statement fired the trigger.

A conditional predicate can appear wherever a BOOLEAN expression can appear.

Example 9-1 creates a DML trigger that uses conditional predicates to determine which of its four possible triggering statements fired it.

Example 9-1 Trigger Uses Conditional Predicates to Detect Triggering Statement

```
CREATE OR REPLACE TRIGGER t
  BEFORE
    INSERT OR
    UPDATE OF salary, department_id OR
    DELETE
  ON employees
BEGIN
  CASE
    WHEN INSERTING THEN
      DBMS_OUTPUT.PUT_LINE('Inserting');
    WHEN UPDATING('salary') THEN
      DBMS_OUTPUT.PUT_LINE('Updating salary');
    WHEN UPDATING('department_id') THEN
      DBMS_OUTPUT.PUT_LINE('Updating department ID');
    WHEN DELETING THEN
      DBMS_OUTPUT.PUT_LINE('Deleting');
  END CASE;
END;
/
```

```
SQL>
SQL> -- create demo table
SQL> create table Employee(
  2    ID              VARCHAR2(4 BYTE)        NOT NULL,
  3    First_Name       VARCHAR2(10 BYTE),
  4    Last_Name        VARCHAR2(10 BYTE),
  5    Start_Date       DATE,
  6    End_Date         DATE,
  7    Salary           Number(8,2),
  8    City             VARCHAR2(10 BYTE),
  9    Description      VARCHAR2(15 BYTE)
```

```
 10  )
 11  /

Table created.

SQL>
SQL> -- prepare data
SQL> insert into Employee(ID,  First_Name, Last_Name, Start_Date,              End_Date,
Salary,  City,       Description)
  2             values ('01','Jason',    'Martin',  to_date('19960725','YYYYMMDD'),
to_date('20060725','YYYYMMDD'), 1234.56, 'Toronto',  'Programmer')
  3  /

1 row created.

SQL> insert into Employee(ID,  First_Name, Last_Name, Start_Date,              End_Date,
Salary,  City,       Description)
  2             values('02','Alison',    'Mathews', to_date('19760321','YYYYMMDD'),
to_date('19860221','YYYYMMDD'), 6661.78, 'Vancouver','Tester')
  3  /

1 row created.

SQL> insert into Employee(ID,  First_Name, Last_Name, Start_Date,              End_Date,
Salary,  City,       Description)
  2             values('03','James',    'Smith',   to_date('19781212','YYYYMMDD'),
to_date('19900315','YYYYMMDD'), 6544.78, 'Vancouver','Tester')
  3  /

1 row created.

SQL> insert into Employee(ID,  First_Name, Last_Name, Start_Date,              End_Date,
Salary,  City,       Description)
  2             values('04','Celia',    'Rice',    to_date('19821024','YYYYMMDD'),
to_date('19990421','YYYYMMDD'), 2344.78, 'Vancouver','Manager')
  3  /

1 row created.

SQL> insert into Employee(ID,  First_Name, Last_Name, Start_Date,              End_Date,
Salary,  City,       Description)
  2             values('05','Robert',   'Black',   to_date('19840115','YYYYMMDD'),
to_date('19980808','YYYYMMDD'), 2334.78, 'Vancouver','Tester')
  3  /

1 row created.

SQL> insert into Employee(ID,  First_Name, Last_Name, Start_Date,              End_Date,
Salary, City,        Description)
  2             values('06','Linda',    'Green',   to_date('19870730','YYYYMMDD'),
to_date('19960104','YYYYMMDD'), 4322.78,'New York',  'Tester')
```

```
  3  /

1 row created.

SQL> insert into Employee(ID,  First_Name, Last_Name, Start_Date,                End_Date,
Salary, City,        Description)
  2              values('07','David',    'Larry',   to_date('19901231','YYYYMMDD'),
to_date('19980212','YYYYMMDD'), 7897.78,'New York',  'Manager')
  3  /

1 row created.

SQL> insert into Employee(ID,  First_Name, Last_Name, Start_Date,                End_Date,
Salary, City,        Description)
  2              values('08','James',    'Cat',     to_date('19960917','YYYYMMDD'),
to_date('20020415','YYYYMMDD'), 1232.78,'Vancouver', 'Tester')
  3  /

1 row created.

SQL>
SQL>
SQL>
SQL> -- display data in the table
SQL> select * from Employee
  2  /

ID   FIRST_NAME          LAST_NAME           START_DAT END_DATE      SALARY CITY
DESCRIPTION
---- ------------------- ------------------- --------- --------- ---------- ---------- ---------------
01   Jason               Martin              25-JUL-96 25-JUL-06    1234.56 Toronto    Programmer
02   Alison              Mathews             21-MAR-76 21-FEB-86    6661.78 Vancouver  Tester
03   James               Smith               12-DEC-78 15-MAR-90    6544.78 Vancouver  Tester
04   Celia               Rice                24-OCT-82 21-APR-99    2344.78 Vancouver  Manager
05   Robert              Black               15-JAN-84 08-AUG-98    2334.78 Vancouver  Tester
06   Linda               Green               30-JUL-87 04-JAN-96    4322.78 New York   Tester
07   David               Larry               31-DEC-90 12-FEB-98    7897.78 New York   Manager
08   James               Cat                 17-SEP-96 15-APR-02    1232.78 Vancouver  Tester

8 rows selected.

SQL>
SQL>
SQL> CREATE OR REPLACE TRIGGER LogRSChanges
  2    BEFORE INSERT OR DELETE OR UPDATE ON employee
  3    FOR EACH ROW
  4  DECLARE
  5    v_ChangeType CHAR(1);
  6  BEGIN
  7    /* Use 'I' for an INSERT, 'D' for DELETE, and 'U' for UPDATE. */
  8    IF INSERTING THEN
```

```
 9     v_ChangeType := 'I';
10   ELSIF UPDATING THEN
11     v_ChangeType := 'U';
12   ELSE
13     v_ChangeType := 'D';
14   END IF;
15
16   DBMS_OUTPUT.put_line(v_ChangeType ||' '|| USER ||' ' ||SYSDATE);
17 END LogRSChanges;
18 /

Trigger created.

SQL>
SQL> update employee set id = '00';
U JAVA2S 07-JUN-07
U JAVA2S 07-JUN-07
U JAVA2S 07-JUN-07
U JAVA2S 07-JUN-07
U JAVA2S 07-JUN-07
U JAVA2S 07-JUN-07
U JAVA2S 07-JUN-07
U JAVA2S 07-JUN-07

8 rows updated.

SQL>
SQL> delete from employee;
D JAVA2S 07-JUN-07
D JAVA2S 07-JUN-07
D JAVA2S 07-JUN-07
D JAVA2S 07-JUN-07
D JAVA2S 07-JUN-07
D JAVA2S 07-JUN-07
D JAVA2S 07-JUN-07
D JAVA2S 07-JUN-07

8 rows deleted.

SQL>
SQL>
SQL>
SQL>
SQL>
SQL> -- clean the table
SQL> drop table Employee
 2 /

Table dropped.
```

Enabling,Disabling and dropping Triggers

how to use the **DROP TRIGGER statement** to drop a trigger in Oracle with syntax and examples.

Description

Once you have created a trigger in Oracle, you might find that you need to remove it from the database. You can do this with the DROP TRIGGER statement.

Syntax

The syntax to a **drop a trigger** in Oracle in Oracle/PLSQL is:

```
DROP TRIGGER trigger_name;
```

Parameters or Arguments

trigger_name

The name of the trigger that you wish to drop.

Example

Let's look at an example of how to drop a trigger in Oracle.

For example:

```
DROP TRIGGER orders_before_insert;
```

This example uses the ALTER TRIGGER statement to drop the trigger called *orders_before_insert*.

Oracle / PLSQL: Disable a Trigger

This Oracle tutorial explains how to **disable a trigger** in Oracle with syntax and examples.

Description

Once you have created a Trigger in Oracle, you might find that you are required to disable the trigger. You can do this with the ALTER TRIGGER statement.

Syntax

The syntax for a disabling a Trigger in Oracle/PLSQL is:

```
ALTER TRIGGER trigger_name DISABLE;
```

Parameters or Arguments

trigger_name

The name of the trigger that you wish to disable.

Example

Let's look at an example that shows how to disable a trigger in Oracle.

For example:

```
ALTER TRIGGER orders_before_insert DISABLE;
```

This example uses the ALTER TRIGGER statement to disable the trigger called *orders_before_insert*.

A disabled trigger can be reenabled .

```
ALTER TRIGGER orders_before_insert ENABLE;
```

Problems on triggers and functions

1. a. Write a trigger that displays a message USER_NAME is performing a transaction on TODAYS_DATE! For example 'RITU is performing a transaction on 11-NOV-2010' (if the username is RITU every time a delete, insert or update is made on table S.

b. Write a trigger that will display this message for every row that is affected by the DML used.

2. Write a trigger called delete_from_s that allows the delete from table S even if there is a foreign key violation, by deleting the corresponding row in table SP first.

3. Write a trigger on table S that inserts the value of SNO automatically into the row while the user inserts the other values for that row.

Sample Input:

SQL>Insert into S(sname,status,city) values ('HARRY',10,'WINDSOR');

1 row inserted

SQL>SELECT * FROM S;

SNO SNAME STATUS CITY

--- ---------- ---------- ----------

S1 SMITH 20 LONDON

S2 JONES 10 PARIS

S3 BLAKE 30 PARIS

S4 CLARK 20 LONDON

S5 ADAMS 30 ATHENS

S6 HARRY 10 WINDSOR

CREATE OR REPLACE TRIGGER PK_ON_S BEFORE INSERT ON S FOR EACH ROW DECLARE A VARCHAR2(20); BEGIN SELECT 'S'||TO_CHAR(MAX (TO_NUMBER(SUBSTR(SNO,2,3)))+1) INTO A FROM S; :NEW.SNO := A; END;

4. a. Write a function called age that takes the date of birth as input and returns the age.

CREATE OR REPLACE FUNCTION age (dob DATE) RETURN NUMBER IS y_old NUMBER(3); BEGIN y_old := MONTHS_BETWEEN(SYSDATE, dob)/12; RETURN y_old; END;

b. Write a trigger called AGE_CHECK on table MODERN_WARFARE that does not allow the insertion or update of any record that has an age less than 18. You must use the function that you created in 4a to calculate the age.

Structure of MODERN_WARFARE is :

Name VARCHAR2(30) DOB DATE

Sample Input:

rituch@cs01>Insert into MODERN_WARFARE values(‘Harry’,'14-Jan-76'); 1 row inserted rituch@cs01> insert into MODERN_WARFARE values('bbb','4-DEC-10'); insert into temp values('bbb','4-DEC-10') * ERROR at line 1: ORA-20012: U must be 18 or older ORA-06512: at "RITUCH.AGE_CHECK", line 3 ORA-04088: error during execution of trigger 'RITUCH.AGE_CHECK'

CREATE OR REPLACE TRIGGER AGE_CHECK BEFORE INSERT ON TEMP FOR EACH ROW BEGIN IF AGE(:NEW.DATE_STORE) < 18 THEN RAISE_APPLICATION_ERROR(-20012,'U MUST BE 18 OR OLDER'); END IF; END;

5. Write a trigger for table S called ensure_case that converts the sname and city to uppercase before they are inserted or updated => if the insert statement given by a user is

Insert into S values (‘S8’, ‘harry’, 20, ‘windsor’) ;

This trigger must convert the name to HARRY and city to WINDSOR before actually inserting them.

CREATE OR REPLACE TRIGGER TEST_TEMP BEFORE INSERT ON TEMP FOR EACH ROW BEGIN :NEW.CHAR_STORE := UPPER(:NEW.CHAR_STORE); END