Harmony IT Solution

Database Design and Programming

Tahaluf Training Center 2022

Harmony IT Solution

1. Conditional control
2. Iterative processing with loops
3. Exception
4. Records
5. Cursors

Harmony IT Solution

IF Statement

- The **IF statement** allows you to either execute or skip a sequence of statements, depending on a condition.

- The IF statement has the three forms:

1. IF THEN
2. IF THEN ELSE
3. IF THEN ELSIF

IF THEN statement:

- The condition is a Boolean expression that always evaluates to TRUE, FALSE, or NULL.
- If the condition evaluates to TRUE, the statements after the THEN execute. Otherwise, the IF statement does nothing.

```
IF condition THEN
    statements;
END IF;
```

Example :

```
--Chapter 8
declare
sales number :=500;
begin
if sales >200 then
Dbms_output.put_Line('sales = '||sales);
End if;
End;
```

IF THEN ELSE statement:

```
IF condition THEN
    statements;
ELSE
    else_statements;
END IF;
```

Example:

```
declare
sales number :=500;
begin
if sales <200 then
Dbms_output.put_Line('sales > '||sales);
else
Dbms_output.put_Line('sales < '||sales);
End if;
End;
```

IF THEN ELSIF statement:

The following illustrates the structure of the IF THEN ELSIF statement:

```
IF condition_1 THEN
  statements_1
ELSIF condition_2 THEN
  statements_2
[ ELSE
    else_statements
]
END IF;
```

Example :

```
declare
sales number :=500;
begin
if sales >600 then
Dbms_output.put_Line('sales > '||sales);
elsif sales<200 then
Dbms_output.put_Line('sales < '||sales);
elsif sales=500 then
Dbms_output.put_Line('sales = '||sales); -- this statement true
End if;
End;
```

Nested IF statement:

You can nest an IF statement within another IF statement as shown below:

```
IF condition_1 THEN
    IF condition_2 THEN
        nested_if_statements;
    END IF;
ELSE
    else_statements;
END IF;
```

Harmony IT Solution

CASE Statement

- The **CASE** statement chooses one sequence of statements to execute out of many possible sequences.

- The CASE statement has two types: **simple CASE statement** and **searched CASE statement**.

- Both types of the CASE statements support an optional **ELSE clause**.

Simple CASE statement:

- A simple **CASE** statement evaluates a single expression and compares the result with some values.

- The simple **CASE** statement has the following structure:

```
CASE selector
WHEN selector_value_1 THEN
   statements_1
WHEN selector_value_2 THEN
   statement_2
ELSE
   else_statements
END CASE;
```

Example :

```
declare
grade CHAR(1);
begin
grade :='B';
case grade -- Simple Case
   WHEN 'A' THEN DBMS_OUTPUT.PUT_LINE('Excellent');
   WHEN 'B' THEN DBMS_OUTPUT.PUT_LINE('Very Good');
   WHEN 'C' THEN DBMS_OUTPUT.PUT_LINE('Good');
   WHEN 'D' THEN DBMS_OUTPUT.PUT_LINE('Fair');
   WHEN 'F' THEN DBMS_OUTPUT.PUT_LINE('Poor');
   ELSE DBMS_OUTPUT.PUT_LINE('No such grade');
 END CASE;
end;
```

Searched CASE statement:

The searched CASE statement evaluates multiple Boolean expressions and executes the sequence of statements associated with the first condition that evaluates to TRUE.

```
CASE
WHEN condition_1 THEN statements_1
WHEN condition_2 THEN statements_2
...
WHEN condition_n THEN statements_n
[ ELSE
  else_statements ]
END CASE;]
```

Example :

```
declare
mark number;
begin
mark :=77;
case --Searched Case
WHEN mark<50  THEN DBMS_OUTPUT.PUT_LINE('F');
WHEN mark >=60 and mark < 70 THEN DBMS_OUTPUT.PUT_LINE('D');
WHEN mark >=70 and mark < 80 THEN DBMS_OUTPUT.PUT_LINE('C');
WHEN mark >=80 and mark < 90 THEN DBMS_OUTPUT.PUT_LINE('B');
WHEN mark >=90 and mark <= 100 THEN DBMS_OUTPUT.PUT_LINE('A');
ELSE DBMS_OUTPUT.PUT_LINE('No such grade');
end case;
end;
```

GOTO statement

- The **GOTO** statement allows you to transfer control to a labeled block or statement.

- The **label_name** is the name of a label that identifies the target statement. In the program, you surround the label name with double enclosing angle brackets as shown below:

```
BEGIN
  GOTO label_1;
  DBMS_OUTPUT.PUT_LINE('This is skipped');
  <<label_1>>
  DBMS_OUTPUT.PUT_LINE('It is here!');
END;
```

```
begin
goto Second_Point;
<<First_Point>>
DBMS_OUTPUT.put_line('First Statement');
goto End_Point;
<<Second_Point>>
DBMS_OUTPUT.put_line('Second Statement');
goto First_Point;
<<End_Point>>
DBMS_OUTPUT.put_line('End Statement');
End;
```

```
Second Statement
First Statement
End Statement
```

Iterative processing with loops

The PL/SQL LOOP statement has the following structure:

```
LOOP
    statements;
END LOOP loop_label;
```

Example :

```
Declare
counter number :=0;
begin
loop
counter := counter+1;
if counter>10 then
Exit;
End IF;
DBMS_OUTPUT.put_line('Counter ='|| counter);
end loop;
DBMS_OUTPUT.put_line('_____');
DBMS_OUTPUT.put_line('Counter ='|| counter);
End;
```

```
Counter =1
Counter =2
Counter =3
Counter =4
Counter =5
Counter =6
Counter =7
Counter =8
Counter =9
Counter =10

Counter =11
```

- PL/SQL **FOR LOOP** executes a sequence of statements a specified number of times.

- The PL/SQL FOR LOOP statement has the following structure:

```
FOR index IN lower_bound .. upper_bound
LOOP
   statements;
END LOOP;
```

Example :

```
Begin
for counter in 1..10
loop
DBMS_OUTPUT.put_line('Counter ='|| counter);
end loop;
end;
```

Example :

```
DECLARE
  g_counter PLS_INTEGER := 10;
BEGIN
  FOR l_counter IN 1.. 5 loop
    DBMS_OUTPUT.PUT_LINE (l_counter);
  end loop;
  -- after the loop
  DBMS_OUTPUT.PUT_LINE (g_counter);
END;
```

Example :

```
Begin --For Loop reverse
for counter in reverse 1..10
loop
DBMS_OUTPUT.put_line('Counter ='|| counter);
end loop;
end;
```

Harmony IT Solution

WHILE loop

- Here is the syntax for the **WHILE LOOP** statement:

```
While Condition
Loop
      Statements;
End Loop;
```

Example :

```
--While Loop
declare
counter number:=0;
begin
DBMS_OUTPUT.put_line('While Loop :');
while counter<10
loop
DBMS_OUTPUT.put_line('Counter ='|| counter);
counter:=counter+1;
end loop;
DBMS_OUTPUT.put_line('_____');
DBMS_OUTPUT.put_line('Counter After While Loop ='|| counter);
end;
```

- The following example is the same as the one above except that it has an additional **EXIT WHEN** statement:

```
--While Loop
declare
counter number:=0;
begin
DBMS_OUTPUT.put_line('While Loop :');
while counter<10
loop
DBMS_OUTPUT.put_line('Counter ='|| counter);
counter:=counter+1;
exit when counter =5;
end loop;
end;
```

CONTINUE statement

- The **CONTINUE** statement allows you to exit the current loop iteration and immediately continue to the next iteration of that loop.

- Typically, the **CONTINUE** statement is used within an IF THEN statement to exit the current loop iteration based on a specified condition as shown below:

```
IF condition THEN
    CONTINUE;
END IF;
```

Example :

```
--Continue Statement
begin
for counter in 1..10
loop
continue when mod(counter,2)=0;
DBMS_OUTPUT.put_line('odd number Counter ='|| counter);
end loop;
end;
```

CONTINUE WHEN statement:

- The **CONTINUE WHEN** statement exits the current loop iteration based on a condition and immediately continue to the next iteration of that loop.

```
CONTINUE WHEN condition;
```

Example :

```
--Continue Statement
begin
for counter in 1..10
loop
exit when mod(counter,2)=0;
DBMS_OUTPUT.put_line('odd number Counter ='|| counter);
end loop;
end;
```

Harmony IT Solution

Exception

- PL/SQL treats all errors that occur in an anonymous block, procedure, or function as **exceptions**.

- The exceptions can have different causes such as coding mistakes, bugs, even hardware failures.

- It is not possible to anticipate all potential exceptions, however, you can write code to handle exceptions to enable the program to continue running as normal.

- The code that you write to handle exceptions is called an **exception handler.**

```
BEGIN
    -- executable section
    ...
    -- exception-handling section
    EXCEPTION
        WHEN e1 THEN
            -- exception_handler1
        WHEN e2 THEN
            -- exception_handler1
        WHEN OTHERS THEN
            -- other_exception_handler
END;
```

Example :

```
--Exception
declare
S_id student.id%type:=5;
S_name student.name%TYPE;
begin
select name ,id into s_name,S_id
from student
where id='A';
DBMS_OUTPUT.put_line('name ='|| S_name);
DBMS_OUTPUT.put_line('ID ='|| S_id);
exception
when no_data_found then
DBMS_OUTPUT.put_line(SQLERRM);
when others then
DBMS_OUTPUT.put_line('invalid input');
end;
```

Example :

Exception	Oracle Error	SQLCODE Value
ACCESS_INTO_NULL	ORA-06530	-6530
CASE_NOT_FOUND	ORA-06592	-6592
COLLECTION_IS_NULL	ORA-06531	-6531
CURSOR_ALREADY_OPEN	ORA-06511	-6511
DUP_VAL_ON_INDEX	ORA-00001	-1
INVALID_CURSOR	ORA-01001	-1001
LOGIN_DENIED	ORA-01017	-1017
NO_DATA_FOUND	ORA-01403	+100

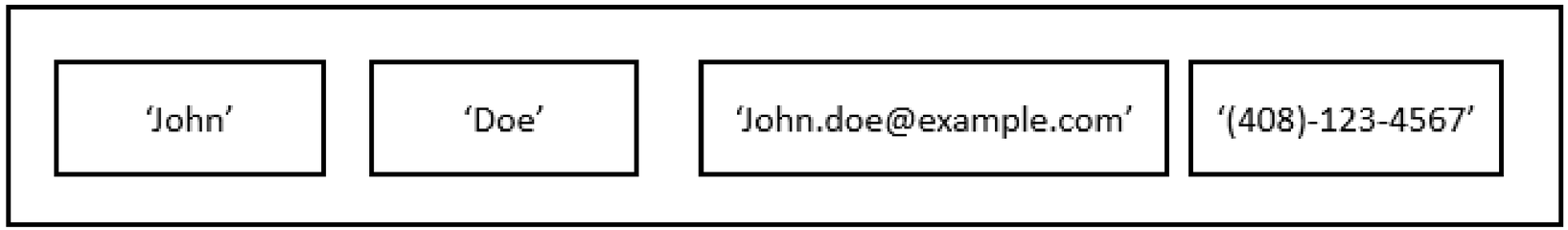
Example :

NOT_LOGGED_ON	ORA-01012	-1012
PROGRAM_ERROR	ORA-06501	-6501
ROWTYPE_MISMATCH	ORA-06504	-6504
SELF_IS_NULL	ORA-30625	-30625
STORAGE_ERROR	ORA-06500	-6500
SUBSCRIPT_BEYOND_COUNT	ORA-06533	-6533
SUBSCRIPT_OUTSIDE_LIMIT	ORA-06532	-6532
SYS_INVALID_ROWID	ORA-01410	-1410
TIMEOUT_ON_RESOURCE	ORA-00051	-51
TOO_MANY_ROWS	ORA-01422	-1422
VALUE_ERROR	ORA-06502	-6502
ZERO_DIVIDE		

Records

- A PL/SQL **record** is a composite data structure which consists of multiple fields; each has its own value.

- The following picture shows an example record that includes **first name**, **last name**, **email**, and **phone number**:



- PL/SQL **record** helps you simplify your code by shifting from field-level to record-level operations.

- PL/SQL has three types of records: **table-based**, **cursor-based**, **programmer-defined**.

Before using a record, you must declare it:

```
declare
record_name table_name%ROWTYPE;
begin
```

Create Table Persons :

```
CREATE TABLE persons  (
    person_id NUMBER GENERATED BY DEFAULT AS IDENTITY,
    first_name VARCHAR2( 50 ) NOT NULL,
    last_name VARCHAR2( 50 ) NOT NULL,
    primary key (person_id)
);
```

Example 1 :

```
declare
r_person persons%rowtype;
begin
--assgin valuse in person record
r_person.person_id:=1;
r_person.first_name:='raya';
r_person.last_name:='hussein';
--insert into person
insert into persons values r_person;
end;
```

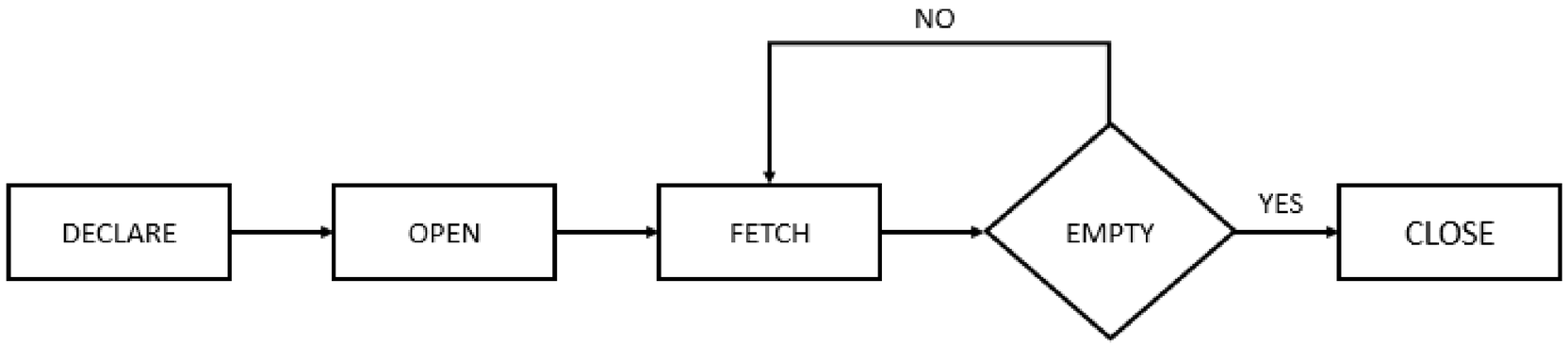
Example 2 :

```
DECLARE
  r_person person%ROWTYPE;
BEGIN
  -- get person data of person id 1
  SELECT * INTO r_person
  FROM person
  WHERE personid = 1;
  -- change the person's last name
  r_person.lastname  := 'Smith';
  -- update the person
  UPDATE person
  SET ROW = r_person
  WHERE personid = r_person.personid;
END;
```

Cursors

- A **cursor** is a pointer to this context area. 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.



- You can name a cursor so that it could be referred to in 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:**

    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.

- The following table provides the description of the most used attributes :

No	Attribute & Description
1	**%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.
2	**%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.
3	**%ISOPEN** Always returns FALSE for implicit cursors, because Oracle closes the SQL cursor automatically after executing its associated SQL statement.
4	**%ROWCOUNT** Returns the number of rows affected by an INSERT, UPDATE, or DELETE statement, or returned by a SELECT INTO statement.

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.

```
CURSOR cursor_name IS select_statement;
```

Working with an explicit cursor includes the following steps:

1. Declaring the cursor for initializing the memory.
2. Opening the cursor for allocating the memory.
3. Fetching the cursor for retrieving the data.
4. Closing the cursor to release the allocated memory.

- **Declaring the Cursor**

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

- **For example :**

```
cursor  c_Employee is
select name,salary from employee ;
```

- **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_employee;
```

- **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_employee into c_name,c_salary;
```

- **Closing the Cursor**

Closing the cursor means releasing the allocated memory.

- **For example, we will close the above-opened cursor as follows :**

```
close c_employee;
```

Example

```
declare
cursor  c_Employee is select name,salary from employee where departmentid=1;
c_name Employee.name%type;
c_salary Employee.salary%type;
BEGIN
 if not c_employee%isopen  then
 open c_employee;
 end if;
 loop
 fetch c_employee into c_name,c_salary;
 exit when c_employee%NotFound;
 DBMS_Output.put_line('name : '|| c_name);
 DBMS_Output.put_line('salary : '|| c_salary);
 end loop;
 DBMS_Output.put_line('Row number : '|| c_employee%rowcount);
 close c_employee;
 Exception
 when invalid_cursor
 then
 DBMS_Output.put_line('Are you missing onen cursor ^_^ !!! ');
END;
```

Harmony IT Solution