

Database development with PL/SQL INSY 8311



ADVENTIST UNIVERSITY
OF CENTRAL AFRICA

Instructor:

- Eric Maniraguha | ericmaniraguha2024@gmail.com | [LinkedIn Profile](#)



6h00 pm – 8h50 pm

- Monday A -G209
- Tuesday B-G209
- Wednesday E-G209
- Thursday F-G308

October 2024

Database development with PL/SQL

Reference reading

- [What is PL/SQL? Explained](#)
- [PL/SQL Exception Handling in All Details](#)



Lecture 07 – Control Structures and Error Handling

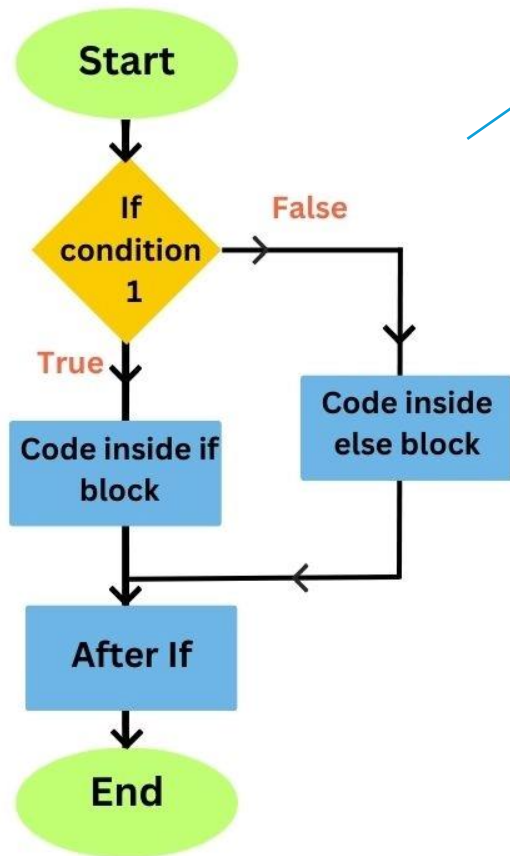
Control Structures and Error Handling in PL/SQL



Control structures and error handling are essential components in PL/SQL that allow developers to control the flow of execution and handle exceptions effectively. In PL/SQL, there are three main types of control structures: **conditional statements**, **iterative statements**, and **sequential control**, along with **error handling mechanisms** to manage exceptions.

Condition Statement

In PL/SQL, conditions are fundamental to controlling the flow of execution. They allow you to execute specific blocks of code based on whether a particular condition is true or false. The two primary types of conditional statements in PL/SQL are **IF statements** and **CASE statements**. Here's a detailed overview of each, along with examples.

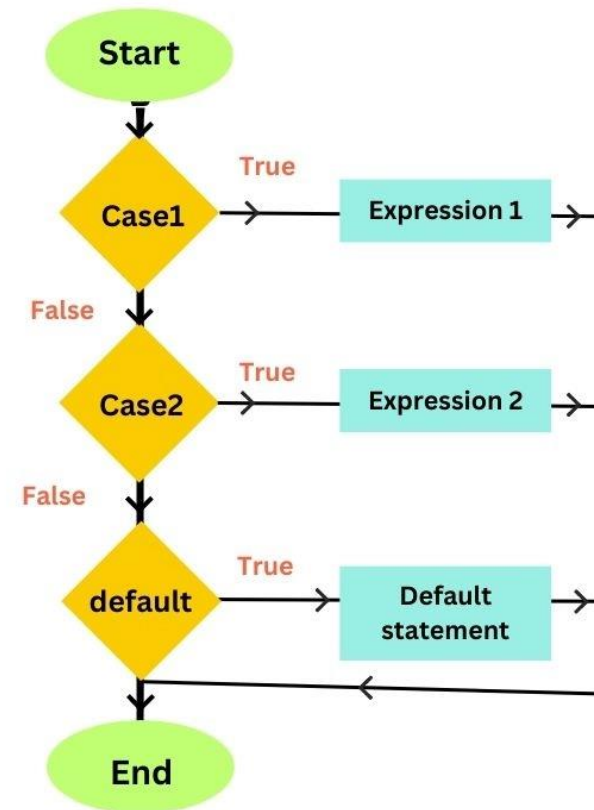


If-Else Diagram:

- The program checks **Condition 1**.
 - If **true**, it runs the **if block**.
 - If **false**, it runs the **else block**.
- After either block runs, the program continues and ends.

Switch-Case Diagram:

- The program checks multiple **cases**.
 - If **Case 1** matches, it runs **Expression 1**.
 - If **Case 2** matches, it runs **Expression 2**.
 - If no case matches, it runs the **default** block.
- The program ends after a match is found or the default runs



IF Statements - Syntax

1. IF-THEN:

```
IF condition THEN
  -- statements to execute if condition is true
END IF;
```

2. IF-THEN-ELSE:

```
IF condition THEN
  -- statements to execute if condition is true
ELSE
  -- statements to execute if condition is false
END IF;
```

3. IF-THEN-ELSIF:

```
IF condition1 THEN
  -- statements for condition1
ELSIF condition2 THEN
  -- statements for condition2
ELSE
  -- statements if none of the conditions are true
END IF;
```

Syntax



```
DECLARE
  score NUMBER := 75; -- Student's score
  grade CHAR(1); -- Variable to hold the grade
BEGIN
  IF score >= 90 THEN
    grade := 'A';
  ELSIF score >= 80 THEN
    grade := 'B';
  ELSIF score >= 70 THEN
    grade := 'C';
  ELSIF score >= 60 THEN
    grade := 'D';
  ELSE
    grade := 'F';
  END IF;

  DBMS_OUTPUT.PUT_LINE('Grade: ' || grade);
END;
/
```

Example Using IF Statement

CASE Statements

The **CASE** statement is useful when you have multiple conditions based on the same expression. It evaluates an expression and executes the corresponding block of code for the first matching value.



1. SIMPLE CASE:

```
CASE expression
  WHEN value1 THEN
    -- statements
  WHEN value2 THEN
    -- statements
  ELSE
    -- statements if no values match
END CASE;
```

2. SEARCH CASE:

```
CASE
  WHEN condition1 THEN
    -- statements
  WHEN condition2 THEN
    -- statements
  ELSE
    -- statements if no conditions match
END CASE;
```

Example Using CASE Statement:

```
DECLARE
  grade CHAR(1) := 'B'; -- Example grade
  message VARCHAR2(100); -- Variable to hold the message
BEGIN
  CASE grade
    WHEN 'A' THEN
      message := 'Excellent work!';
    WHEN 'B' THEN
      message := 'Good job!';
    WHEN 'C' THEN
      message := 'Fair performance.';
    WHEN 'D' THEN
      message := 'Needs improvement.';
    WHEN 'F' THEN
      message := 'Failed. Please try again.';
    ELSE
      message := 'Invalid grade.';
  END CASE;


  DBMS_OUTPUT.PUT_LINE('Message: ' || message);
END;
/
```


If Statement and Case Statement



Summary of Conditions





 **IF Statements:** Good for evaluating a single condition or multiple conditions using **ELSIF**. They are versatile and can be nested.

 **CASE Statements:** More concise when you have multiple conditions based on the same expression. They improve readability when checking the same variable against different values.

Best Practices:



 Use **IF** statements for complex conditions and scenarios where multiple conditions need to be checked.

 Use **CASE** statements when you have multiple values to compare against a single expression for better readability

Best practices for conditional statements:

- Use indentation and alignment for readability.
- Clarify complex conditions with parentheses.
- Prefer CASE over nested IF statements for multiple alternatives.
- Handle null values using NULLIF and COALESCE functions.



PL/SQL Control Structures (1/4)

A. Conditional Statements (anonymous PL/SQL block)

- Conditional statements allow the program to make decisions based on certain conditions.
- IF-THEN Statement: Executes a set of statements if a condition is true.

```
DECLARE
    v_salary NUMBER := 3000; -- Initialize the variable 'v_salary'
    with a value of 3000
BEGIN
    -- Check if the salary is greater than 2000
    IF v_salary > 2000 THEN
        DBMS_OUTPUT.PUT_LINE('High Salary'); -- Output
        message indicating a high salary
    END IF; -- End of the IF statement
END;
/
```

```
DECLARE
    v_salary NUMBER := 1500; -- Initialize the variable 'v_salary' with a value
    of 1500
BEGIN
    -- Check if the salary is greater than 2000
    IF v_salary > 2000 THEN
        DBMS_OUTPUT.PUT_LINE('High Salary'); -- Output message indicating
        a high salary
    ELSE
        DBMS_OUTPUT.PUT_LINE('Low Salary'); -- Output message indicating a
        low salary
    END IF; -- End of the IF statement
END;
/
```

B. IF-THEN-ELSE Statement: Executes one set of statements if a condition is true and another if it is false.

PL/SQL Control Structures (2/4)

C. IF-THEN-ELSIF-ELSE Statement: Allows multiple conditions to be tested in sequence.

```
DECLARE
    v_salary NUMBER := 1500; -- Initialize the variable 'v_salary'
    with a value of 1500
BEGIN
    -- Check if the salary is greater than 2000
    IF v_salary > 2000 THEN
        DBMS_OUTPUT.PUT_LINE('High Salary'); -- Output
        message for high salary
    ELSIF v_salary BETWEEN 1000 AND 2000 THEN
        DBMS_OUTPUT.PUT_LINE('Moderate Salary'); -- Output
        message for moderate salary
    ELSE
        DBMS_OUTPUT.PUT_LINE('Low Salary'); -- Output
        message for low salary
    END IF; -- End of the IF statement
END;
/
```

```
DECLARE
    v_grade CHAR(1) := 'B'; -- Initialize the variable 'v_grade' with a value of
    'B'
BEGIN
    -- Evaluate the value of 'v_grade' using a CASE statement
    CASE v_grade
        WHEN 'A' THEN
            DBMS_OUTPUT.PUT_LINE('Excellent'); -- Output message for grade
            A
        WHEN 'B' THEN
            DBMS_OUTPUT.PUT_LINE('Good'); -- Output message for grade B
        WHEN 'C' THEN
            DBMS_OUTPUT.PUT_LINE('Average'); -- Output message for grade C
        ELSE
            DBMS_OUTPUT.PUT_LINE('Fail'); -- Output message for any grade
            not A, B, or C
    END CASE; -- End of the CASE statement
END;
/
```

D. CASE Statement: A simplified version of multiple IF conditions.

PL/SQL Control Structures (3/4)

E. Iterative Statements

- Iterative statements allow the program to loop over a set of statements.
- **LOOP Statement:** Executes a set of statements repeatedly until explicitly exited using EXIT.

```
DECLARE
    i NUMBER := 1; -- Declare and initialize a variable 'i' with the
value 1
BEGIN
    -- Start the loop
    LOOP
        -- Display the current iteration number
        DBMS_OUTPUT.PUT_LINE('Iteration ' || i);

        -- Increment the value of 'i' by 1
        i := i + 1;

        -- Exit the loop when 'i' becomes greater than 5
        EXIT WHEN i > 5;
    END LOOP;
END;
```

```
DECLARE
    v_grade CHAR(1) := 'B'; -- Initialize the variable 'v_grade' with a value of
'B'
BEGIN
    -- Evaluate the value of 'v_grade' using a CASE statement
    CASE v_grade
        WHEN 'A' THEN
            DBMS_OUTPUT.PUT_LINE('Excellent'); -- Output message for grade
A
        WHEN 'B' THEN
            DBMS_OUTPUT.PUT_LINE('Good'); -- Output message for grade B
        WHEN 'C' THEN
            DBMS_OUTPUT.PUT_LINE('Average'); -- Output message for grade C
        ELSE
            DBMS_OUTPUT.PUT_LINE('Fail'); -- Output message for any grade
not A, B, or C
    END CASE; -- End of the CASE statement
END;
/
```

F. WHILE LOOP: Repeats a set of statements as long as a condition is true.

PL/SQL Control Structures (4/4)

G. FOR LOOP: Executes a set of statements a fixed number of times.

```
DECLARE
  i NUMBER(1); -- Declare variable 'i' of type NUMBER to be
                -- used in the outer loop
  j NUMBER(1); -- Declare variable 'j' of type NUMBER to be
                -- used in the inner loop
BEGIN
  << outer_loop >> -- Label for the outer loop
  FOR i IN 1..3 LOOP -- Outer loop that iterates from 1 to 3
    << inner_loop >> -- Label for the inner loop
    FOR j IN 1..3 LOOP -- Inner loop that iterates from 1 to 3
      dbms_output.put_line('i is: ' || i || ' and j is: ' || j); -- Print
the values of 'i' and 'j' in each iteration
    END LOOP inner_loop; -- End of the inner loop
  END LOOP outer_loop; -- End of the outer loop
END;
/
```

```
DECLARE
  v_count NUMBER := 0; -- Initialize a variable 'v_count' with value 0
BEGIN
  GOTO skip; -- Jump to the 'skip' label and bypass the following line
  v_count := v_count + 1; -- This line is skipped due to the GOTO
statement

  <<skip>> -- Label where execution resumes after the GOTO statement
  DBMS_OUTPUT.PUT_LINE('Count: ' || v_count); -- Output the current
value of 'v_count'
END;
/
```

H. Sequential Control

Sequential control statements allow the use of control structures like **GOTO** to jump to specific labels within a program.

GOTO Statement: Directs program flow to a specified label.

Loops in PL/SQL

In PL/SQL, loops are used to repeatedly execute a block of code as long as a certain condition is true or for a specified number of iterations. There are three types of loops in PL/SQL: **simple loops**, **WHILE loops**, and **FOR loops**. Each type serves a different purpose depending on the nature of the task.



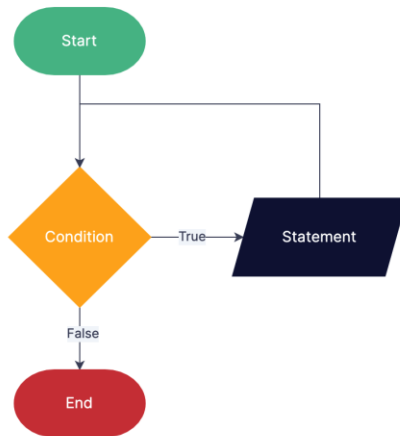
Key Differences Between Loop Types

Loop Type	Use Case	Condition Check	Control of Iterations
Simple Loop	Repeated execution until an EXIT condition	Inside the loop body	Requires manual EXIT
WHILE Loop	When you want to repeat code as long as a condition is true	Before entering the loop	Condition checked before each iteration
FOR Loop	When you know how many times the loop should run	Implicitly within bounds	Automatically controlled by range

Loops in PL/SQL

A **WHILE loop** executes the block of code as long as the specified condition is true. The condition is evaluated before each iteration, meaning the loop may never execute if the condition is initially false.

While Loop Flowchart



```

DECLARE
  i NUMBER := 1; -- Declare a variable 'i' of type NUMBER and initialize it to 1
BEGIN
  WHILE i <= 5 LOOP -- Start a WHILE loop that will iterate as long as 'i' is less
    DBMS_OUTPUT.PUT_LINE('Iteration ' || i); -- Print the current iteration number
    using 'i'
    i := i + 1; -- Increment 'i' by 1 after each iteration
  END LOOP; -- End of the WHILE loop
END;
/
  
```

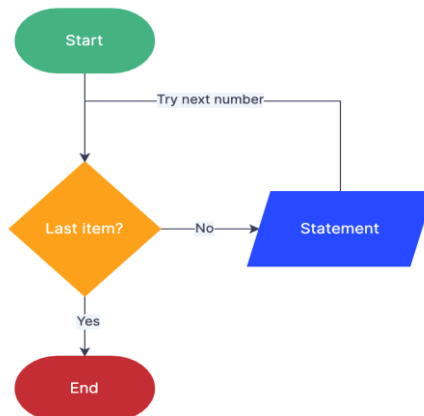
Output



Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5

A **FOR loop** is used when the number of iterations is known beforehand. It automatically increments the loop variable and terminates when the condition is met. A FOR loop does not require an EXIT statement as it knows when to stop.

For Loop Flowchart



```

BEGIN
  FOR i IN 1..5 LOOP -- A FOR loop that runs from 1 to 5, inclusive
    DBMS_OUTPUT.PUT_LINE('Iteration ' || i); -- Print the current iteration number
    using 'i'
  END LOOP; -- End of the FOR loop
END;
/
  
```

Output:
Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5

=====Reversed=====

```

BEGIN
  FOR i IN REVERSE 1..5 LOOP
    DBMS_OUTPUT.PUT_LINE('Iteration ' || i);
  END LOOP;
END;
/
  
```



Output-Reversed:

Iteration 5
Iteration 4
Iteration 3
Iteration 2
Iteration 1

Nested Loop Demonstration: Counting with Limits



```
DECLARE
  a NUMBER := 0; -- Initialize variable 'a' with 0
  b NUMBER; -- Declare variable 'b'
  upper_limit NUMBER := 4; -- Set the upper limit for the outer loop
BEGIN
  dbms_output.put_line('Program started.');
```

-- Display starting message

```

  <<outer_loop>> -- Label for the outer loop
  LOOP
    a := a + 1; -- Increment 'a' by 1
    b := a; -- Initialize 'b' with the value of 'a'

    <<inner_loop>> -- Label for the inner loop
    LOOP
      EXIT outer_loop WHEN a > upper_limit; -- Exit the outer loop when 'a' exceeds the upper limit
      dbms_output.put_line('a = ' || a); -- Print the value of 'a'

      b := b + 1; -- Increment 'b' by 1
      EXIT inner_loop WHEN b > a; -- Exit the inner loop when 'b' exceeds 'a'
    END LOOP inner_loop;
  END LOOP outer_loop;

  dbms_output.put_line('Program completed.');
```

-- Display completion message

```
END;
/
```

Output:

```
Program started.
a = 1
a = 2
a = 3
a = 4
Program completed.
```

PL/SQL procedure successfully completed.

```
DECLARE
  a NUMBER := 0; -- Initialize variable 'a' with 0
  b NUMBER; -- Declare variable 'b'
  upper_limit NUMBER := 4; -- Set the upper limit for the outer loop
BEGIN
  dbms_output.put_line('Program started.');
```

-- Display starting message

```

  LOOP -- Outer loop (no label needed)
    a := a + 1; -- Increment 'a' by 1
    b := a; -- Initialize 'b' with the value of 'a'

    LOOP -- Inner loop (no label needed)
      EXIT WHEN a > upper_limit; -- Exit the outer loop when 'a' exceeds the upper limit
      dbms_output.put_line('a = ' || a); -- Print the value of 'a'

      b := b + 1; -- Increment 'b' by 1
      EXIT WHEN b > a; -- Exit the inner loop when 'b' exceeds 'a'
    END LOOP;
  END LOOP;

  dbms_output.put_line('Program completed.');
```

-- Display completion message

```
END;
/
```

Conclusion:

- Labels are optional and primarily useful for nested loops when you need to manage specific loops with EXIT or CONTINUE.
- For simple loops, labels are unnecessary.

While Loop vs For Loop – Examples

```
DECLARE
-- No need to declare 'x' explicitly, as 'FOR' loop handles the variable
declaration automatically.
BEGIN
-- Start a 'FOR' loop, where 'x' ranges from 1 to 10.
FOR x IN 1..10 LOOP
-- Output the current value of 'x' using DBMS_OUTPUT.PUT_LINE.
DBMS_OUTPUT.PUT_LINE('x = ' || x);
END LOOP; -- End of the loop, after 10 iterations.
END;
/
```

First Block: Using FOR Loop

```
DECLARE
-- Declare a variable 'x' of type NUMBER and initialize it with the value 1
x NUMBER := 1;
BEGIN
-- Start the WHILE loop, which will continue as long as x is less than or
equal to 10
WHILE x <= 10 LOOP
-- Output the current value of 'x' using DBMS_OUTPUT.PUT_LINE
DBMS_OUTPUT.PUT_LINE('x = ' || x);

-- Increment 'x' by 1
x := x + 1;
END LOOP; -- End of the loop when the condition is no longer true (x >
10)
END;
/
```

First Block: Using WHILE Loop

Key Differences While Loop vs For Loop

Criteria	WHILE Loop	FOR Loop
Control	Runs as long as the condition is true	Runs for a specific, predefined number of times
Iterations	Unknown, can change dynamically	Known and fixed, determined at the start
Use Case	Complex conditions, possibly dynamic iteration	Simple, predictable range iteration
Management of Index	Must be managed manually inside the loop	Handled automatically by the loop
Loop Structure	Flexible, can exit in the middle or change logic	Simple, predefined range, less flexibility

Summary:

- **Choose WHILE:** If you want more flexibility and the loop may need to continue or break based on complex or dynamic conditions.
- **Choose FOR:** When you know exactly how many times the loop should run or want to iterate over a range of values (e.g., 1 to 10).

If you don't need dynamic or complex control and the loop iterations are predictable, the FOR loop is typically preferred for simplicity and readability.

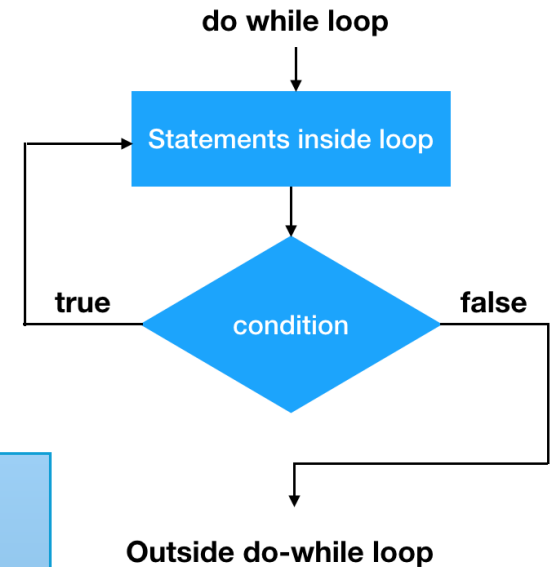
While and Do-While

In PL/SQL, a **DO WHILE** loop isn't directly available, but you can achieve similar functionality using a **WHILE** loop. Here's an example that simulates a **DO WHILE** loop by ensuring that the body of the loop is executed at least once:



```
DECLARE
  v_counter NUMBER := 1; -- Initialize a counter variable
BEGIN
  -- Use a WHILE loop to simulate a DO WHILE loop
  LOOP
    DBMS_OUTPUT.PUT_LINE('Counter: ' || v_counter); -- Print the current counter value
    v_counter := v_counter + 1; -- Increment the counter

    -- Exit condition to stop the loop
    EXIT WHEN v_counter > 5; -- Exit when the counter exceeds 5
  END LOOP; -- End of the loop
END;
/
```



Output:

Counter: 1
Counter: 2
Counter: 3
Counter: 4
Counter: 5

PL/SQL procedure successfully completed.

Loop to drop multiple users dynamically



```
DECLARE
  -- Define a collection (VARRAY) of usernames to drop
  TYPE username_array IS VARRAY(10) OF VARCHAR2(30);
  usernames username_array := username_array('eric', 'student', 'admin', 'pdbadmin','oracle');
BEGIN
  -- Loop through the collection
  FOR i IN 1 .. usernames.COUNT LOOP
    BEGIN
      -- Attempt to drop the user
      EXECUTE IMMEDIATE 'DROP USER ' || usernames(i) || ' CASCADE';
      DBMS_OUTPUT.PUT_LINE('User ' || usernames(i) || ' dropped successfully.');
```

```
EXCEPTION
  WHEN OTHERS THEN
    -- Handle the error if the user doesn't exist or other issues arise
    DBMS_OUTPUT.PUT_LINE('Error dropping user ' || usernames(i) || ': ' || SQLERRM);
  END;
END LOOP;
END;
```

Transformation of the anonymous block into a PL/SQL stored procedure – FOR LOOP

This will loop through numbers from 1 to 10, printing the value of x during each iteration, and you will be able to call the procedure after creation.



```
DECLARE
  -- Declare a variable 'x' of type NUMBER and initialize it with the
  value 1
  x NUMBER := 1;
BEGIN
  -- Start the loop
  LOOP
    -- Output the current value of 'x' using DBMS_OUTPUT.PUT_LINE
    DBMS_OUTPUT.PUT_LINE('x = ' || x);

    -- Increment 'x' by 1
    x := x + 1;

    -- Exit the loop when 'x' is greater than 10
    EXIT WHEN x > 10;
  END LOOP;
END;
/
```

Anonymous Block

```
-- Create a procedure to loop through numbers 1 to 10
CREATE OR REPLACE PROCEDURE print_loop IS
  -- Declare the variable x
  x NUMBER := 1;
BEGIN
  -- Start the loop
  LOOP
    -- Print the current value of x
    DBMS_OUTPUT.PUT_LINE('x = ' || x);

    -- Increment x by 1
    x := x + 1;

    -- Exit the loop when x exceeds 10
    EXIT WHEN x > 10;
  END LOOP;
END;
/

-- Call the procedure
BEGIN
  print_loop;
END;
/
```

Store Procedure

Transformation of the anonymous block into a PL/SQL stored procedure – WHILE LOOP

This will loop through numbers from 1 to 10, printing the value of x during each iteration, and you will be able to call the procedure after creation.



```
DECLARE
  -- Declare a variable 'x' of type NUMBER and initialize it with the
  value 1
  x NUMBER := 1;
BEGIN
  -- Start the WHILE loop, which will continue as long as x is less than
  or equal to 10
  WHILE x <= 10 LOOP
    -- Output the current value of 'x' using DBMS_OUTPUT.PUT_LINE
    DBMS_OUTPUT.PUT_LINE('x = ' || x);

    -- Increment 'x' by 1
    x := x + 1;
  END LOOP;
END;
/
```

Anonymous Block

```
-- Create a procedure to print numbers from 1 to 10 using
a WHILE loop
CREATE OR REPLACE PROCEDURE print_numbers IS
  -- Declare a variable 'x' of type NUMBER and initialize it
  with the value 1
  x NUMBER := 1;
BEGIN
  -- Start the WHILE loop, which will continue as long as x
  is less than or equal to 10
  WHILE x <= 10 LOOP
    -- Output the current value of 'x' using
    DBMS_OUTPUT.PUT_LINE
    DBMS_OUTPUT.PUT_LINE('x = ' || x);

    -- Increment 'x' by 1
    x := x + 1;
  END LOOP;
END;
/

-- Call the procedure
BEGIN
  print_numbers;
END;
/
```

Store Procedure

Exception or Error Handling in PL/SQL

In PL/SQL, **exception or error handling** is done using the EXCEPTION section in a PL/SQL block. **Exceptions are runtime errors or warnings that occur during program execution.** PL/SQL provides a mechanism **to catch and handle exceptions gracefully.**



```
DECLARE
  -- Declare variables and types here
BEGIN
  -- Main executable code
  -- Example: Some SQL or PL/SQL code that might raise an exception
EXCEPTION
  -- Exception handling code
  WHEN <exception_name1> THEN
    -- Action to take if exception_name1 occurs
  WHEN <exception_name2> THEN
    -- Action to take if exception_name2 occurs
  WHEN OTHERS THEN
    -- Action to take for any other exception (catch-all)
END;
```

Structure of a PL/SQL Block with Exception Handling

Type of Exception

Predefined Exceptions: These are exceptions that are automatically raised by PL/SQL whenever a particular error condition occurs. For example:

NO_DATA_FOUND: Raised when a SELECT INTO statement returns no rows.

TOO_MANY_ROWS: Raised when a SELECT INTO statement returns more than one row.

ZERO_DIVIDE: Raised when a division by zero is attempted.

INVALID_CURSOR: Raised when an operation is performed on an invalid cursor.

User-Defined Exceptions: You can define your own exceptions using the EXCEPTION keyword and raise them with the RAISE statement.

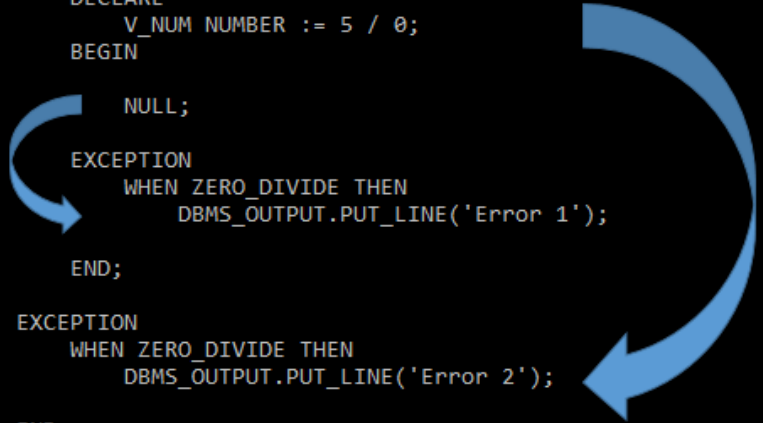
Commonly Used Predefined Exceptions in PL/SQL

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
INVALID_NUMBER	ORA-01722	-1722
LOGIN_DENIED	ORA-01017	-1017
NO_DATA_FOUND	ORA-01403	+100
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

```

SQL> SET SERVEROUTPUT ON;
SQL> BEGIN
2
3   DECLARE
4       V_NUM NUMBER := 5 / 0;
5   BEGIN
6
7       NULL;
8
9   EXCEPTION
10      WHEN ZERO_DIVIDE THEN
11          DBMS_OUTPUT.PUT_LINE('Error 1');
12
13  END;
14
15 EXCEPTION
16      WHEN ZERO_DIVIDE THEN
17          DBMS_OUTPUT.PUT_LINE('Error 2');
18
19 END;
20 /
Error 2
PL/SQL yordam² bama²yla tamamland².

```



Exception	Oracle Error	SQLCODE Value
SUBSCRIPT_OUTSIDE_LIMIT	ORA-06532	-6532
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	ORA-01476	-1476

Exception or Error Handling in PL/SQL - Examples



```
DECLARE
  l_num NUMBER; -- Declare a variable to hold the salary
BEGIN
  -- Attempt to select salary for employee with ID 1001
  -- This SELECT INTO statement can raise exceptions like
  NO_DATA_FOUND or TOO_MANY_ROWS
  SELECT salary INTO l_num FROM employees WHERE employee_id =
  1001;

  EXCEPTION
    -- Handle the case where no employee is found with the specified ID
    WHEN NO_DATA_FOUND THEN
      DBMS_OUTPUT.PUT_LINE('No employee found with ID 1001');

    -- Handle the case where multiple employees are found with the same
    ID
    WHEN TOO_MANY_ROWS THEN
      DBMS_OUTPUT.PUT_LINE('More than one employee found with the
      same ID');
  END;
```

Example with Predefined Exception

```
DECLARE
  e_invalid_salary EXCEPTION; -- Declare a user-defined exception
  for invalid salary
  l_salary NUMBER; -- Declare a variable to hold the salary
BEGIN
  l_salary := -5000; -- Assign an invalid salary value
  -- Check if the salary is invalid (less than 0), and raise the custom
  exception if true
  IF l_salary < 0 THEN
    RAISE e_invalid_salary; -- Raise the user-defined exception for
    invalid salary
  END IF;
  EXCEPTION
    -- Handle the user-defined invalid salary exception
    WHEN e_invalid_salary THEN
      DBMS_OUTPUT.PUT_LINE('Invalid salary detected.');
```

Example with User-Defined Exception

Catch-All Exception in PL/SQL – WHEN OTHERS

The **Catch-All Exception** in PL/SQL is handled using the WHEN OTHERS clause in the EXCEPTION block. This allows you to catch and handle any exceptions that are not explicitly handled by other WHEN clauses. The WHEN OTHERS exception is useful when you want to capture unexpected errors or when you don't need to handle each possible error individually.

Key Points about WHEN OTHERS:

It catches all exceptions that are not handled by any specific WHEN clause.

It must be the last exception handler in the EXCEPTION block, as no further exception clauses will be evaluated once it is encountered.

It is often used in conjunction with SQLCODE and SQLERRM to log or display detailed error information.

```
BEGIN
  -- Main executable block
EXCEPTION
  WHEN OTHERS THEN
    -- Handle any exceptions that are not
    explicitly handled
END;
```

Structure of WHEN OTHERS

Catch-All Exception in PL/SQL – WHEN OTHERS/Example

```
BEGIN
  -- Attempt to divide by zero (this will raise a ZERO_DIVIDE exception)
  DECLARE
    num1 NUMBER := 10;
    num2 NUMBER := 0;
    result NUMBER;
  BEGIN
    result := num1 / num2; -- This line will raise an exception
  END;
EXCEPTION
  -- Catch all unhandled exceptions (like division by zero)
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('An unexpected error occurred.');
```

-- Optionally log the error code and message

```
    DBMS_OUTPUT.PUT_LINE('Error Code: ' || SQLCODE);
    DBMS_OUTPUT.PUT_LINE('Error Message: ' || SQLERRM);
END;
```

Best Practices:

Use sparingly: Only use WHEN OTHERS when you cannot predict all possible exceptions or when you're handling them in a generic way (e.g., logging them for later debugging).

Log errors: Always try to log the error code and message, especially when using WHEN OTHERS, so that you can diagnose the issue later.

Avoid silencing errors: Avoid using WHEN OTHERS without some form of error logging, as it could hide potential bugs.

This is how **WHEN OTHERS** can serve as a catch-all for unforeseen exceptions, ensuring your PL/SQL code can handle unexpected runtime errors gracefully.

Example of Using WHEN OTHERS

Exception Handling – Extra Examples (1/2)

```
DECLARE
    v_name VARCHAR2(50); -- Declare a variable 'v_name' of type
    VARCHAR2 to store the employee's first name
BEGIN
    SELECT first_name INTO v_name -- Select the 'first_name' from the
    'employees' table and store it in 'v_name'
    FROM employees
    WHERE employee_id = 9999; -- The query looks for an employee
    with 'employee_id' 9999
EXCEPTION
    WHEN NO_DATA_FOUND THEN -- Handle the exception if no row is
    returned by the query
        DBMS_OUTPUT.PUT_LINE('No employee found with that ID.');
```

Print an error message when no data is found

```
END;
/
```

Predefined Exceptions

These are exceptions that are automatically raised by PL/SQL when certain conditions occur.

User-Defined Exceptions

You can define your own exceptions to handle specific error cases.

```
DECLARE
    v_salary NUMBER := 4000; -- Declare a variable 'v_salary' of type
    NUMBER and initialize it to 4000
    e_high_salary EXCEPTION; -- Declare a custom exception
    'e_high_salary'
BEGIN
    IF v_salary > 3000 THEN -- Check if 'v_salary' is greater than 3000
        RAISE e_high_salary; -- Raise the custom exception
    'e_high_salary'
    END IF;
EXCEPTION
    WHEN e_high_salary THEN -- Handle the custom exception
    'e_high_salary'
        DBMS_OUTPUT.PUT_LINE('Salary exceeds the limit.');
```

Print the message 'Salary exceeds the limit.'

```
END;
/
```

Exception Handling – Extra Examples (2/2)

```
DECLARE
  v_salary NUMBER := 4000; -- Declare a variable 'v_salary' of type NUMBER and initialize it to 4000
BEGIN
  IF v_salary > 3000 THEN -- Check if 'v_salary' is greater than 3000
    RAISE_APPLICATION_ERROR(-20001, 'Salary exceeds the limit.');
```

-- Raise an application error with code -20001 and a custom error message

```
  END IF; -- End of the IF condition
END;
/
```

RAISE_APPLICATION_ERROR

This function allows you to create custom error messages.

PL/SQL Procedure to Calculate Tax for a Purchase

Procedure

```
CREATE OR REPLACE PROCEDURE calculate_tax (  
    p_purchase_amount IN NUMBER, -- Input: Purchase amount  
    p_tax_rate        IN NUMBER, -- Input: Tax rate as a percentage (e.g., 10 for 10%)  
    p_tax_amount      OUT NUMBER -- Output: Calculated tax amount  
) IS  
BEGIN  
    -- Check if purchase amount or tax rate is negative  
    IF p_purchase_amount < 0 THEN  
        DBMS_OUTPUT.PUT_LINE('Error: Purchase amount cannot be negative.');        p_tax_amount := 0; -- Set tax amount to 0 if invalid  
    ELSIF p_tax_rate < 0 THEN  
        DBMS_OUTPUT.PUT_LINE('Error: Tax rate cannot be negative.');        p_tax_amount := 0; -- Set tax amount to 0 if invalid  
    ELSE  
        -- Calculate the tax based on the purchase amount and tax rate  
        p_tax_amount := (p_purchase_amount * p_tax_rate) / 100;  
    END IF;  
EXCEPTION  
    -- Handle any unexpected errors  
    WHEN OTHERS THEN  
        DBMS_OUTPUT.PUT_LINE('An unexpected error occurred while calculating tax.');        DBMS_OUTPUT.PUT_LINE('Error Code: ' || SQLCODE);  
        DBMS_OUTPUT.PUT_LINE('Error Message: ' || SQLERRM);  
        p_tax_amount := 0; -- Set tax amount to 0 in case of an error  
END calculate_tax;
```

```
DECLARE  
    v_tax_amount NUMBER; -- Variable to hold the calculated tax  
                           amount  
BEGIN  
    -- Call the procedure and pass the purchase amount and tax rate  
    calculate_tax(500, 10, v_tax_amount); -- 500 is the purchase  
                                           amount, 10% is the tax rate  
  
    -- Output the calculated tax amount  
    DBMS_OUTPUT.PUT_LINE('The calculated tax is: ' ||  
        v_tax_amount);  
END;
```

Example of Calling the Procedure

Conclusion: Key Takeaways -Error Handling

1. EXCEPTION for Transaction Handling

Write a procedure *usp_TransferFunds* that transfers money from one account to another. If there is an error during the transfer, rollback the transaction and print an error message.

2. Nested Error Handling

Write a procedure *usp_ProcessPayment* that calculates the payment amount and processes it. Use nested BEGIN-EXCEPTION-END blocks to handle different error types (e.g., insufficient funds, invalid account).

3. RAISE_APPLICATION_ERROR with Conditions

Write a procedure *usp_OrderValidation* that validates an order by checking the stock. If the stock is insufficient, raise a custom error with a descriptive message.

4. EXCEPTION for Input Validation

Create a procedure *usp_InputValidation* that validates input data for an employee (e.g., age, salary, name). Raise errors for invalid data and use EXCEPTION to handle any errors that occur.

5. EXCEPTION for Null Handling

Write a procedure *usp_CalculateBonus* that accepts an employee ID and calculates the bonus. Handle null values in the employee's salary and raise an error if the salary is null.

6. Using EXCEPTION: INVALID_NUMBER

Create a procedure *usp_AddTwoNumbers* that accepts two parameters and adds them. Implement error handling to catch INVALID_NUMBER if non-numeric data is passed.

7. Catching TOO_MANY_ROWS Exception

Write a procedure *usp_FindSingleEmployee* that queries for an employee by last name. Use an exception to handle cases where more than one employee with the same last name exists (i.e., TOO_MANY_ROWS).

8. Catching Custom Errors

Write a procedure *usp_PlaceOrder* that accepts an order ID and product ID. If the order quantity exceeds 100, raise a custom exception. Use the OTHERS exception handler to catch all other unexpected errors.

Conclusion: Key Takeaways – Control Structure

1. Basic IF-THEN-ELSE Logic

Write a procedure *usp_CalculateDiscount* that accepts an *order_amount* and applies a discount based on the following rules:

- If the order amount is greater than 5000, apply a 10% discount.
- If the order amount is between 3000 and 5000, apply a 5% discount.
- Otherwise, no discount is applied.
- The procedure should print the final amount after the discount.



2. LOOP with EXIT Condition

Write a procedure *usp_DisplayNumbers* that uses a LOOP to print numbers from 1 to 10. Use an EXIT WHEN condition to stop the loop after 10 iterations.

3. WHILE LOOP Practice

Write a procedure *usp_Countdown* that accepts a number and prints a countdown from that number to 1 using a WHILE LOOP.

5. Combining Control Structures

Write a procedure *usp_CalculateShippingFee* that calculates the shipping fee based on the weight of a package. Use the following rules:

- If the weight is less than 1kg, the fee is \$5.
- If the weight is between 1kg and 5kg, the fee is \$10.
- If the weight is above 5kg, charge \$20.
- Use an IF-THEN-ELSE block for this logic.

6. Nested IF-THEN-ELSE

Create a procedure *usp_TaxCalculator* that calculates the tax based on the income:

- If income is greater than 100,000, tax rate is 30%.
- If income is between 50,000 and 100,000, tax rate is 20%.
- If income is below 50,000, tax rate is 10%. Use nested IF-THEN statements to calculate the tax and print the tax amount.

Keep Learning!