# PL/SQL

Advantages of PL/SQL

PL/SQL has the following advantages −

- SQL is the standard database language and PL/SQL is strongly integrated with SQL. PL/SQL supports both static and dynamic SQL.
- PL/SQL allows sending an entire block of statements to the database at one time. This reduces network traffic and provides high performance for the applications.
- PL/SQL gives high productivity to programmers as it can query, transform, and update data in a database.
- PL/SQL saves time on design and debugging by strong features, such as exception handling, encapsulation, data hiding, and object-oriented data types.
- Applications written in PL/SQL are fully portable.
- PL/SQL provides high security level.
- PL/SQL provides access to predefined SQL packages.
- PL/SQL provides support for Object-Oriented Programming.
- PL/SQL provides support for developing Web Applications and Server Pages.

## PL/SQL Block Structure

- DECLARE (optional)
  - Variables, cursors, user-defined exceptions
- BEGIN (mandatory)
  - SQL statements
  - PL/SQL statements
- EXCEPTION (optional)
  - Actions to perform when errors occur
- END; (mandatory)


- **Variables and Constants**

PL/SQL lets you declare constants and variables, then use them in SQL and procedural statements anywhere an expression can be used.

### Declaring Variables

- Variables can have any SQL datatype, such as CHAR, DATE, or NUMBER, or any PL/SQL datatype, such as BOOLEAN or BINARY_INTEGER.

```
part_no  NUMBER(4);
in_stock BOOLEAN;
```

### Assigning Values to a Variable

- **The first way** uses the assignment operator (:=), a colon followed by an equal sign. You place the variable to the left of the operator and an expression (which can include function calls) to the right. A few examples follow:

```
tax := price * tax_rate;
valid_id := FALSE;
```

- **The second way** to assign values to a variable is by selecting (or fetching) database values into it.
- In the example below, you have Oracle compute a 10% bonus when you select the salary of an employee. Now, you can use the variable bonus in another computation or insert its value into a database table.

SELECT sal * 0.10 INTO bonus FROM emp WHERE empno = emp_id;

- **The third way** to assign values to a variable is by passing it as an OUT or IN OUT parameter to a subprogram. As the following example shows, an OUT parameter lets you initialize sum vaiable by addition of a and b variable.

PROCEDURE add(a number,b number, sum out number)

## Declaring Constants

Declaring a constant is like declaring a variable except that you must add the keyword CONSTANT and immediately assign a value to the constant. Thereafter, no more assignments to the constant are allowed. In the following example, you declare a constant named credit_limit:

credit_limit CONSTANT REAL := 5000.00;

## Logical Operators

The logical operators `AND`, `OR`, and `NOT` follow the tri-state logic shown in Table 2-2. `AND` and `OR` are binary operators; `NOT` is a unary operator.

*Table 2-2 Logic Truth Table*

| **X** | **y** | **x AND y** | **x OR y** | **NOT x** |
|---|---|---|---|---|
| TRUE | TRUE | TRUE | TRUE | FALSE |

| | | | | |
|---|---|---|---|---|
| TRUE | FALSE | FALSE | TRUE | FALSE |
| TRUE | NULL | NULL | TRUE | FALSE |
| FALSE | TRUE | FALSE | TRUE | TRUE |
| FALSE | FALSE | FALSE | FALSE | TRUE |
| FALSE | NULL | FALSE | NULL | TRUE |
| NULL | TRUE | NULL | TRUE | NULL |
| NULL | FALSE | FALSE | NULL | NULL |
| NULL | NULL | NULL | NULL | NULL |

## Boolean Expressions

- **PL/SQL lets you compare variables and constants in both SQL and procedural statements. These comparisons, called *Boolean expressions***, consist of simple or complex expressions separated by relational operators.
- Often, Boolean expressions are **connected by the logical operators `AND`, `OR`, and `NOT`**.
- A Boolean expression **always yields `TRUE`, `FALSE`, or `NULL`.**
- There are three kinds of Boolean expressions: arithmetic, character, and date.

## Boolean Arithmetic Expressions

- You can use the relational operators to **compare numbers** for equality or inequality.
- Comparisons are quantitative; that is, one number is greater than another if it represents a larger quantity. For example, given the assignments

  ```
  number1 := 75;
  number2 := 70;
  ```

  the following expression is true:

  ```
  number1 > number2
  ```

## Boolean Character Expressions

- You can **compare character values** for equality or inequality.
- By default, comparisons are based on the binary values of each byte in the string.

  For example, given the assignments

  ```
  string1 := 'Kathy';
  string2 := 'Kathleen';
  ```

  the following expression is true:

string1 > string2

### Boolean Date Expressions

- You can also **compare dates**.
- Comparisons are chronological; that is, one date is greater than another if it is more recent. For example, given the assignments

  date1 := '01-JAN-91';
  date2 := '31-DEC-90';

  the following expression is true:

  date1 > date2

## CASE Expressions

- A `CASE` expression selects a result from one or more alternatives, and returns the result.
- The `CASE` expression uses a **selector**, an expression whose value determines which alternative to return.
- A `CASE` expression has the following form:

```
CASE selector
  WHEN expression1 THEN result1
  WHEN expression2 THEN result2
  ...
  WHEN expressionN THEN resultN
 [ELSE resultN+1]
```

```
END;
```

- The selector is followed by one or more `WHEN` clauses, which are checked sequentially.
- The value of the selector determines which clause is executed.
- The first `WHEN` clause that matches the value of the selector determines the result value, and subsequent `WHEN` clauses are not evaluated.
- An example follows:

```
D
ECLARE
  grade CHAR(1) := 'B';
  appraisal VARCHAR2(20);
BEGIN
  appraisal :=
    CASE grade
      WHEN 'A' THEN 'Excellent'
      WHEN 'B' THEN 'Very Good'
      WHEN 'C' THEN 'Good'
      WHEN 'D' THEN 'Fair'
      WHEN 'F' THEN 'Poor'
      ELSE 'No such grade'
    END;
END;
```

The optional `ELSE` clause works similarly to the `ELSE` clause in an `IF` statement.If the value of the selector is not one of the choices covered by a `WHEN` clause, the `ELSE` clause is executed. If no `ELSE` clause is provided and none of the `WHEN` clauses are matched, the expression returns `NULL`.

## Searched CASE Expression

- PL/SQL also provides a *searched* CASE expression, which has the form:

```
CASE
   WHEN search_condition1 THEN result1
   WHEN search_condition2 THEN result2
   ...
   WHEN search_conditionN THEN resultN
  [ELSE resultN+1]
END;
```

- **A searched CASE expression has no selector.**
- Each WHEN clause contains a search condition that yields a Boolean value, which lets you test different variables or multiple conditions in a single WHEN clause.
- An example follows:

```
DECLARE
   grade CHAR(1);
   appraisal VARCHAR2(20);
BEGIN
   ...
   appraisal :=
      CASE
         WHEN grade = 'A' THEN 'Excellent'
         WHEN grade = 'B' THEN 'Very Good'
         WHEN grade = 'C' THEN 'Good'
         WHEN grade = 'D' THEN 'Fair'
         WHEN grade = 'F' THEN 'Poor'
         ELSE 'No such grade'
      END;
```

```
    ...
END;
```

- The search conditions are evaluated sequentially.
- The Boolean value of each search condition determines which WHEN clause is executed.
- If a search condition yields TRUE, its WHEN clause is executed. After any WHEN clause is executed, subsequent search conditions are not evaluated.
- If none of the search conditions yields TRUE, the optional ELSE clause is executed.
- If no WHEN clause is executed and no ELSE clause is supplied, the value of the expression is NULL.

## Handling Null Values in Comparisons and Conditional Statements

When working with nulls, you can avoid some common mistakes by keeping in mind the following rules:

- Comparisons involving nulls always yield NULL
- Applying the logical operator NOT to a null yields NULL

- In conditional control statements, if the condition yields NULL, its associated sequence of statements is not executed
- If the expression in a simple CASE statement or CASE expression yields NULL, it cannot be matched by using WHEN NULL. In this case, you would need to use the searched case syntax and test WHEN *expression* IS NULL.
- In the example below, you might expect the sequence of statements to execute because x and y seem unequal. But, nulls are indeterminate. Whether or not x is equal to y is unknown. Therefore, the IF condition yields NULL and the sequence of statements is bypassed.

```
x := 5;
y := NULL;
...
IF x != y THEN  -- yields NULL, not TRUE
   sequence_of_statements;  -- not executed
END IF;
```

- In the next example, you might expect the sequence of statements to execute because a and b seem equal. But, again, that is unknown, so the IF condition yields NULL and the sequence of statements is bypassed.

```
a := NULL;
b := NULL;
...
IF a = b THEN  -- yields NULL, not TRUE
   sequence_of_statements;  -- not executed
END IF;
```

## NOT Operator

- Applying the logical operator NOT to a null yields NULL.

- Thus, the following two statements are not always equivalent:

```
IF x > y THEN     |     IF NOT x > y THEN
   high := x;     |        high := y;
ELSE              |     ELSE
   high := y;     |        high := x;
END IF;           |     END IF;
```

- The sequence of statements in the ELSE clause is executed when the IF condition yields FALSE or NULL. If neither x nor y is null, both IF statements assign the same value to high. However, if either x or y is null, the first IF statement assigns the value of y to high, but the second IF statement assigns the value of x to high.

**Zero-Length Strings**

- **PL/SQL treats any zero-length string like a null**.
- This includes values returned by character functions and Boolean expressions.
- For example, the following statements assign nulls to the target variables:

```
my_string := TO_CHAR('');
```

- So, use the **IS NULL** operator to test for null strings, as follows:

```
IF my_string IS NULL THEN ...
```

**Concatenation Operator**

- **The concatenation operator ignores null operands.**
- For example, the expression

```
'apple' || NULL || NULL || 'sauce'
```

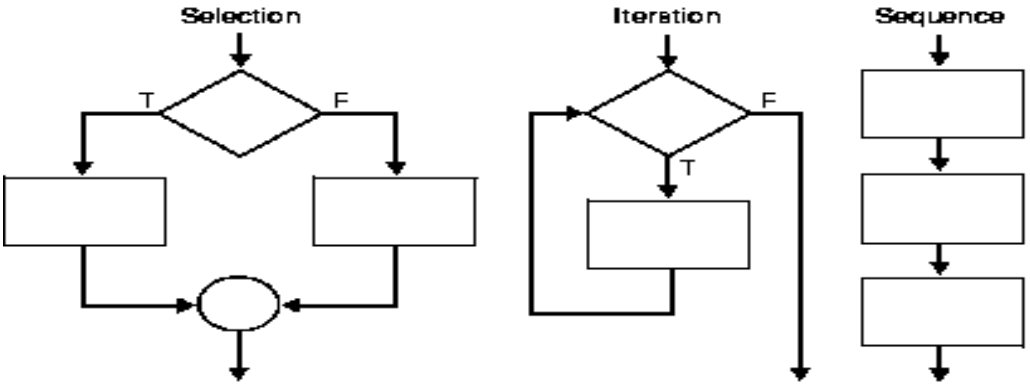
returns the following value:

'applesauce'

## Overview of PL/SQL Control Structures

According to the *structure theorem*, any computer program can be written using the basic control structures shown in Figure 4-1. They can be combined in any way necessary to deal with a given problem.

*Figure 4-1 Control Structures*




- The selection structure tests a condition, then executes one sequence of statements instead of another, depending on whether the condition is true or false. A *condition* is any variable or expression that returns a Boolean value (`TRUE` or `FALSE`).
- The iteration structure executes a sequence of statements repeatedly as long as a condition holds true.
- The sequence structure simply executes a sequence of statements in the order in which they occur.

## Conditional Control: IF and CASE Statements

- `IF` statement lets you execute a sequence of statements conditionally. That is, whether the sequence is executed or not depends on the value of a condition.
- There are three forms of `IF` statements: `IF-THEN`, `IF-THEN-ELSE`, and `IF-THEN-ELSIF`.
- The **`CASE` statement is a compact way** to evaluate a single condition and choose between many alternative actions.

## IF-THEN Statement

- The simplest form of `IF` statement associates a condition with a sequence of statements enclosed by the keywords `THEN` and `END IF` (not `ENDIF`), as follows:

```
IF condition THEN
  sequence_of_statements
END IF;
```

- The sequence of statements is executed only if the condition is true. If the condition is false or null, the `IF` statement does nothing. In either case, control passes to the next statement. An example follows:

```
IF x > y THEN
 high := x;
END IF;
```

## IF-THEN-ELSE Statement

- The second form of `IF` statement adds the keyword `ELSE` followed by an alternative sequence of statements, as follows:

```
IF condition THEN
  sequence_of_statements1;
ELSE
  sequence_of_statements2;
END IF;
```

- The sequence of statements in the `ELSE` clause is executed only if the condition is false or null.
- Thus, the `ELSE` clause ensures that a sequence of statements is executed.
- In the following example, the first `UPDATE` statement is executed when the condition is true, but the second `UPDATE` statement is executed when the condition is false or null:

```
IF trans_type = 'CR' THEN
  UPDATE accounts SET balance = balance + credit
WHERE ...
ELSE
  UPDATE accounts SET balance = balance - debit
WHERE ...
END IF;
```

## IF-THEN-ELSIF Statement

- Sometimes you want to select an action from several mutually exclusive alternatives. The third form of `IF` statement uses the keyword `ELSIF` (not `ELSEIF`) to introduce additional conditions, as follows:

```
IF condition1 THEN
```

```
  sequence_of_statements1
ELSIF condition2 THEN
  sequence_of_statements2
ELSE
  sequence_of_statements3
END IF;
```

- If the first condition is false or null, the `ELSIF` clause tests another condition.
- An `IF` statement can have any number of `ELSIF` clauses; the final `ELSE` clause is optional.
- Conditions are evaluated one by one from top to bottom.
- If any condition is true, its associated sequence of statements is executed and control passes to the next statement.
- If all conditions are false or null, the sequence in the `ELSE` clause is executed. Consider the following example:

```
BEGIN
  ...
  IF sales > 50000 THEN
    bonus := 1500;
  ELSIF sales > 35000 THEN
    bonus := 500;
  ELSE
    bonus := 100;
  END IF;
  INSERT INTO payroll VALUES (emp_id, bonus, ...);
END;
```

- If the value of `sales` is larger than 50000, the first and second conditions are true. Nevertheless, `bonus` is assigned the proper value of 1500 because the second condition is never tested. When the first condition is true, its associated statement is executed and control passes to the `INSERT` statement.

## Iterative Control: LOOP and EXIT Statements

`LOOP` statements let you execute a sequence of statements multiple times. There are three forms of `LOOP` statements: `LOOP`, `WHILE-LOOP`, and `FOR-LOOP`.

### LOOP

- The simplest form of `LOOP` statement is the basic (or infinite) loop, which encloses a sequence of statements between the keywords `LOOP` and `END LOOP`, as follows:

```
LOOP
  sequence_of_statements
END LOOP;
```

- With each iteration of the loop, the sequence of statements is executed, then control resumes at the top of the loop.

- You can use an `EXIT` statement to complete the loop. You can place one or more `EXIT` statements anywhere inside a loop, but nowhere outside a loop.

- There are two forms of `EXIT` statements: **`EXIT` and `EXIT-WHEN`**.

### EXIT

- The `EXIT` statement forces a loop to complete unconditionally.
- When an `EXIT` statement is encountered, the loop completes immediately and control passes to the next statement.
- An example follows:

```
LOOP
  ...
  IF credit_rating < 3 THEN
    ...
    EXIT;  -- exit loop immediately
  END IF;
END LOOP;
-- control resumes here
```

## EXIT-WHEN

- The `EXIT-WHEN` statement lets a loop complete conditionally.
- When the `EXIT` statement is encountered, the condition in the `WHEN` clause is evaluated. If the condition is true, the loop completes and control passes to the next statement after the loop.
- An example follows:

```
LOOP
  ……
  EXIT WHEN i>=10;  -- exit loop if condition is true
  ...
END LOOP;
```

- **Until the condition is true, the loop cannot complete**. So, a statement inside the loop must change the value of the condition.

- In the above example, ifthe value of i is less than 10, the condition is false. When the `value of i becomes equal to or greater than 10`, the condition is true, the loop completes, and control passes to the `CLOSE` statement.
- The `EXIT-WHEN` statement replaces a simple `IF` statement. For example, compare the following statements:

| EXIT | EXIT…WHEN |
|---|---|
| IF count > 100 THEN<br>…….<br>EXIT;<br>END IF; | EXIT WHEN count > 100; |

- These statements are logically equivalent, but the `EXIT-WHEN` statement is easier to read and understand.

## Loop Labels

- Loops can be labeled. The label, an undeclared identifier enclosed by double angle brackets, must appear at the beginning of the `LOOP` statement, as follows:

```
<<label_name>>
LOOP
  sequence_of_statements
END LOOP;
```

- Optionally**, the label name can also appear at the end of the `LOOP` statement**, as the following example shows:

```
<<my_loop>>
LOOP
  ...
END LOOP my_loop;
```

## WHILE-LOOP

- The `WHILE-LOOP` statement associates a condition with a sequence of statements enclosed by the keywords `LOOP` and `END LOOP`, as follows:

```
WHILE condition
 LOOP
   sequence_of_statements
END LOOP;
```

- Before each iteration of the loop, the condition is evaluated.
- **If the condition is true,** the sequence of statements is executed, then control resumes at the top of the loop.
- **If the condition is false or null,** the loop is bypassed and control passes to the next statement.
- An example follows:

```
WHILE total <= 25000
LOOP
  ...
  SELECT sal INTO salary FROM emp WHERE ...
  total := total + salary;
END LOOP;
```

## FOR-LOOP

- Whereas the number of iterations through a `WHILE` loop is unknown until the loop completes, the number of iterations through a `FOR` loop is known before the loop is entered.
- `FOR` loops iterate over a specified range of integers.
- The range is part of an ***iteration scheme*****,** which is enclosed by the keywords `FOR` and `LOOP`. A double dot (`..`) serves as the range operator. The syntax follows:

  ```
  FOR counter IN [REVERSE] lower_bound..higher_bound
  LOOP
    sequence_of_statements
  END LOOP;
  ```

- The range is evaluated when the `FOR` loop is first entered and is never re-evaluated.

As the next example shows, the sequence of statements is executed once for each integer in the range. After each iteration, the loop counter is incremented.

```
FOR i IN 1..3 LOOP  -- assign the values 1,2,3 to i
  sequence_of_statements  -- executes three times
END LOOP;
```

The following example shows that if the lower bound equals the higher bound, the sequence of statements is executed once:

```
FOR i IN 3..3 LOOP  -- assign the value 3 to i
  sequence_of_statements  -- executes one time
END LOOP;
```

By default, iteration proceeds upward from the lower bound to the higher bound. However, as the example below shows, if you use the keyword `REVERSE`, iteration proceeds downward from the higher bound to the lower bound. After each iteration, the loop counter is decremented. Nevertheless, you write the range bounds in ascending (not descending) order.

```
FOR i IN REVERSE 1..3 LOOP  -- assign the values 3,2,1 to i
  sequence_of_statements  -- executes three times
END LOOP;
```

## Using the EXIT Statement

The `EXIT` statement lets a `FOR` loop complete prematurely. For example, the following loop normally executes ten times, but as soon as the `FETCH` statement fails to return a row, the loop completes no matter how many times it has executed:

```
FOR j IN 1..10 LOOP
  FETCH c1 INTO emp_rec;
  EXIT WHEN c1%NOTFOUND;
  ...
END LOOP;
```

Suppose you must exit from a nested `FOR` loop prematurely. You can complete not only the current loop, but any enclosing loop. Simply label the enclosing loop that you want to complete. Then, use the label in an `EXIT` statement to specify which `FOR` loop to exit, as follows:

```
<<outer>>
FOR i IN 1..5 LOOP
  ...
  FOR j IN 1..10 LOOP
    FETCH c1 INTO emp_rec;
    EXIT outer WHEN c1%NOTFOUND;  -- exit both FOR loops
    ...
  END LOOP;
END LOOP outer;
```

## Sequential Control: GOTO and NULL Statements

### GOTO Statement

The `GOTO` statement branches to a label unconditionally. The label must be unique within its scope and must precede an executable statement or a PL/SQL block. When executed, the `GOTO` statement transfers control to the labeled statement or block. In the following example, you go to an executable statement farther down in a sequence of statements:

```
BEGIN
  ...
  GOTO insert_row;
  ...
  <<insert_row>>
  INSERT INTO emp VALUES ...
END;
```

In the next example, you go to a PL/SQL block farther up in a sequence of statements:

```
BEGIN
```

```
  ...
  <<update_row>>
  BEGIN
    UPDATE emp SET ...
    ...
  END;
  ...
  GOTO update_row;
  ...
END;
```

The label `end_loop` in the following example is not allowed because it does not precede an executable statement:

```
DECLARE
  done  BOOLEAN;
BEGIN
  ...
  FOR i IN 1..50 LOOP
    IF done THEN
      GOTO end_loop;
    END IF;
    ...
  <<end_loop>>  -- not allowed
  END LOOP;  -- not an executable statement
END;
```

To debug the last example, just add the `NULL` statement, as follows:

```
FOR i IN 1..50 LOOP
  IF done THEN
    GOTO end_loop;
  END IF;
  ...
<<end_loop>>
```

```
NULL;  -- an executable statement
END LOOP;
```

As the following example shows, a `GOTO` statement can branch to an enclosing block from the current block:

```
DECLARE
  my_ename  CHAR(10);
BEGIN
  <<get_name>>
  SELECT ename INTO my_ename FROM emp WHERE ...
  BEGIN
    ...
    GOTO get_name;  -- branch to enclosing block
  END;
END;
```

The `GOTO` statement branches to the first enclosing block in which the referenced label appears.

## NULL Statement

The `NULL` statement does nothing other than pass control to the next statement. In a conditional construct, the `NULL` statement tells readers that a possibility has been considered, but no action is necessary.

I In the following example, the `NULL` statement emphasizes that only top-rated employees get bonuses:

```
IF rating > 90 THEN
  compute_bonus(emp_id);
ELSE
  NULL;
```

```
END IF;
```

Also, the `NULL` statement is a handy way to create stubs when designing applications from the top down.