PL/SQL:
concept of Stored Procedures &
Functions, Cursors, Triggers,
Assertions, roles and privileges , Embedded SQL, Dynamic SQL

PL/SQL

- **PL/SQL (Procedural Language/Structured Query Language)** is Oracle Corporation's procedural extension for SQL and the Oracle relational database. PL/SQL is available in Oracle Database.

- PL/SQL is a combination of SQL along with the procedural features of programming languages.

- It was developed by Oracle Corporation in the early 90's to enhance the capabilities of SQL.

- PL/SQL is one of three key programming languages embedded in the Oracle Database, along with SQL itself and Java.

- PL/SQL includes procedural language elements such as conditions and loops. It allows declaration of constants and variables, procedures and functions.

- It can handle exceptions (runtime errors). Arrays are supported involving the use of PL/SQL collections.

- Implementations from version 8 of Oracle Database onwards have included features associated with object-orientation. One can create PL/SQL units such as procedures, functions, packages, types, and triggers, which are stored in the database.

❑ Feaures of PL/SQL −

- PL/SQL is a completely portable, high-performance transaction-processing language.
- PL/SQL provides a built-in, interpreted and OS independent programming environment.
- PL/SQL can also directly be called from the command-line **SQL*Plus interface**.
- PL/SQL's general syntax is based on that of ADA and Pascal programming language.
- PL/SQL is tightly integrated with SQL.
- It offers extensive error checking.
- It offers numerous data types.

- It offers a variety of programming structures.
- It supports structured programming through functions and procedures.
- It supports object-oriented programming.
- It supports the development of web applications and server pages.

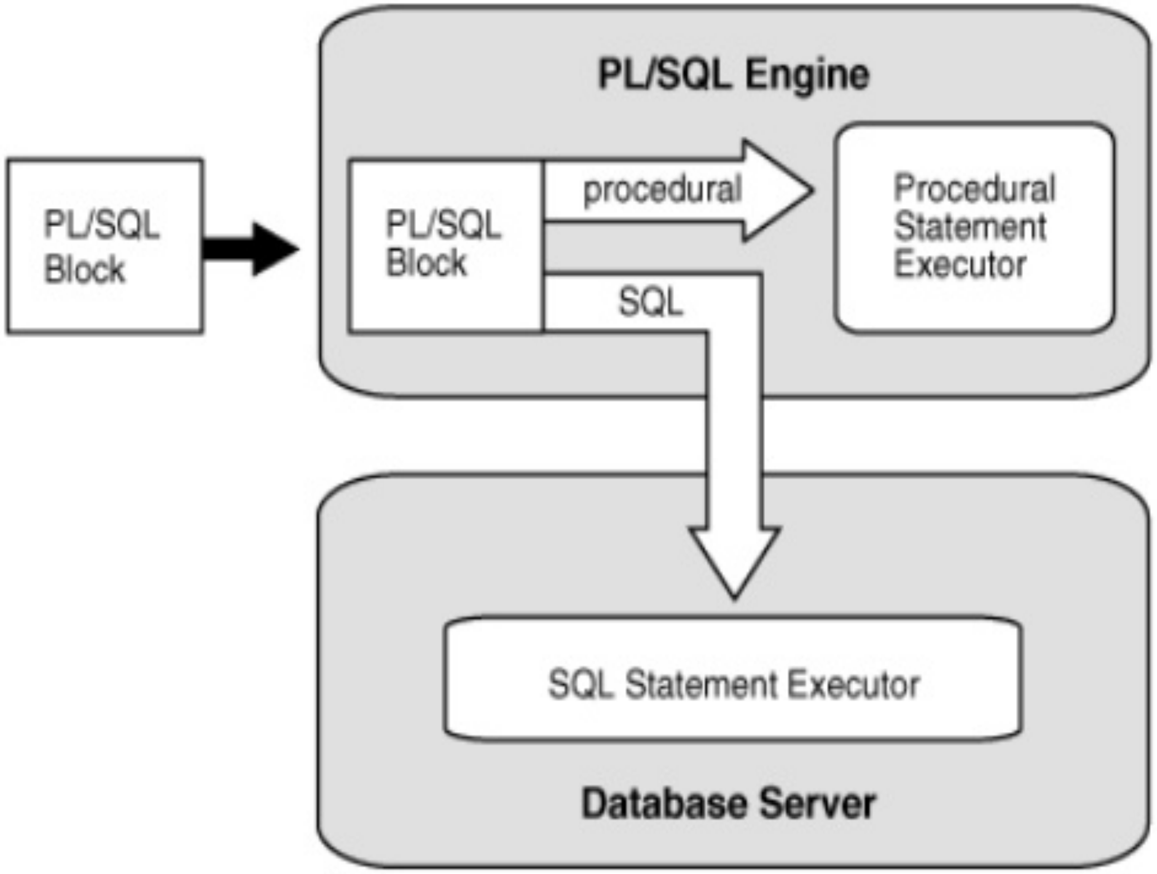
What is PL/SQL?

- **PL/SQL** basically **stands** for "Procedural Language extensions to SQL".
- This is the extension of Structured Query Language (SQL) that is used in Oracle. PL/SQL allows the programmer to write code in procedural format.
- It combines the data manipulation power of SQL with the processing power of procedural language to create a super powerful SQL queries.
- It allows the programmers to instruct the compiler 'what to do' through SQL and 'how to do' through its procedural way.
- Similar to other database languages, it gives more control to the programmers by the use of loops, conditions and object oriented concepts.

Architecture of PL/SQL

- **The PL/SQL architecture mainly consists of following 3 components:**
- PL/SQL block
- PL/SQL Engine
- Database Server

PL/SQL Architecture



PL/SQL block

- This is the component which has the actual PL/SQL code.
- This consists of different sections to divide the code logically (declarative section for declaring purpose, execution section for processing statements, exception handling section for handling errors)
- It also contains the SQL instruction that used to interact with the database server.
- All the PL/SQL units are treated as PL/SQL blocks, and this is the starting stage of the architecture which serves as the primary input.
- Following are the different type of PL/SQL units.
  - Anonymous Block
  - Function
  - Library
  - Procedure
  - Package Body
  - Package Specification
  - Trigger
  - Type
  - Type Body

PL/SQL Engine

- PL/SQL engine is the component where the actual processing of the codes takes place.
- PL/SQL engine separates PL/SQL units and SQL part in the input.
- The separated PL/SQL units will be handled with the PL/SQL engine itself.
- The SQL part will be sent to database server where the actual interaction with database takes place.
- It can be installed in both database server and in the application server.

Database Server

- This is the most important component of PL/SQL unit which stores the data.

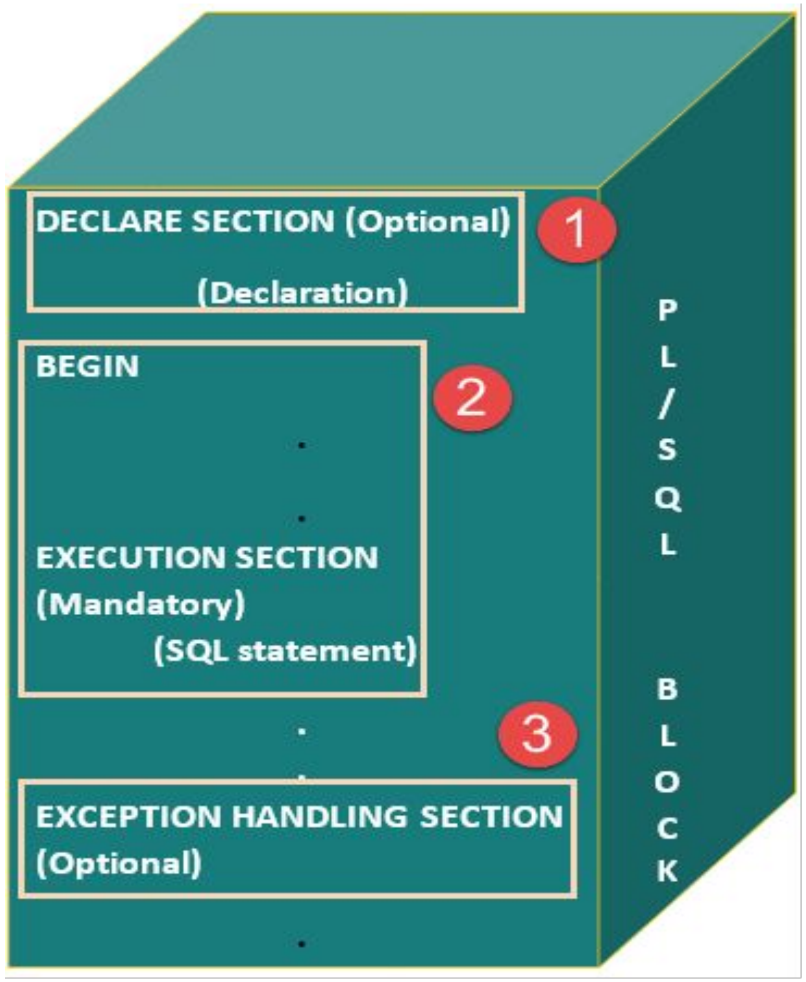
- The PL/SQL engine uses the SQL from PL/SQL units to interact with the database server.

- It consists of SQL executor which actually parses the input SQL statements and execute the same.

Basic Difference between SQL and PL/SQL

SQL	PL/SQL
•SQL is a single query that is used to perform DML and DDL operations.	•PL/SQL is a block of codes that used to write the entire program blocks/ procedure/ function, etc.
•It is declarative, that defines what needs to be done, rather than how things need to be done.	•PL/SQL is procedural that defines how the things needs to be done.
•Execute as a single statement.	•Execute as a whole block.
•Mainly used to manipulate data.	•Mainly used to create an application.
•Interaction with Database server.	•No interaction with the database server.
•Cannot contain PL/SQL code in it.	•It is an extension of SQL, so it can contain SQL inside it.

- Basic Syntax of PL/SQL which is a **block-structured** language; this means that the PL/SQL programs are divided and written in logical blocks of code. Each block consists of three sub-parts −


DECLARE SECTION (Optional)
(Declaration)
1
BEGIN
2
EXECUTION SECTION (Mandatory)
(SQL statement)
3
EXCEPTION HANDLING SECTION (Optional)
PL/SQL BLOCK

S.No	Sections & Description
1	**Declarations** This section starts with the keyword **DECLARE**. It is an optional section and defines all variables, cursors, subprograms, and other elements to be used in the program.
2	**Executable Commands** This section is enclosed between the keywords **BEGIN** and **END** and it is a mandatory section. It consists of the executable PL/SQL statements of the program. It should have at least one executable line of code, which may be just a **NULL command** to indicate that nothing should be executed.
3	**Exception Handling** This section starts with the keyword **EXCEPTION**. This optional section contains **exception(s)** that handle errors in the program.

✔ Declaration Section

- This is the first section of the PL/SQL blocks.

- This is the section in which the declaration of variables, cursors, exceptions, subprograms, pragma instructions and collections that are needed in the block will be declared.

- This particular section is optional and can be skipped if no declarations are needed.

- This should be the first section in a PL/SQL block, if present.

- This section should be always followed by execution section.

- This section starts with the keyword ‘**DECLARE**’.

✔ Execution Section

- Execution part is the main and mandatory part which actually executes the code that is written inside it.

- Since the PL/SQL expects the executable statements from this block this cannot be an empty block, i.e., it should have at least one valid executable code line in it.

- This can contain both PL/SQL code and SQL code.

- This can contain one or many blocks inside it as a nested blocks.

- This section starts with the keyword 'BEGIN'.

- This section should be followed either by 'END' or Exception-Handling section (if present)

✔ Exception-Handling Section:

- The exception are unavoidable in the program which occurs at run-time and to handle this Oracle has provided an Exception-handling section in blocks. This section can also contain PL/SQL statements. This is an optional section of the PL/SQL blocks.

- This is the section where the exception raised in the execution block is handled.

- This section is the last part of the PL/SQL block.

- Control from this section can never return to the execution block.

- This section starts with the keyword **'EXCEPTION'.**

- This section should be always followed by the keyword 'END'.

- The Keyword 'END' marks the end of PL/SQL block.

- Every PL/SQL statement ends with a semicolon (;). PL/SQL blocks can be nested within other PL/SQL blocks using **BEGIN** and **END**.

- Following is the basic structure of a PL/SQL block –

```
DECLARE
    <declarations section>
    BEGIN
        <executable command(s)>
    EXCEPTION
        <exception handling>
END;
```

- **The 'Hello World' Example**

```
DECLARE
    message varchar2(20):= 'Hello, World!';
BEGIN
    dbms_output.put_line(message);
END;
/
```

- The **END;** line signals the end of the PL/SQL block. To run the code from the SQL command line, you may need to type / at the beginning of the first blank line after the last line of the code.

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

```
Hello World
 PL/SQL procedure successfully completed.
```

Advantages of PL/SQL

- SQL is the standard database language and PL/SQL is strongly integrated with SQL.
- PL/SQL supports both static and dynamic SQL.
- Static SQL supports DML operations and transaction control from PL/SQL block.
- In Dynamic SQL, SQL allows embedding DDL statements in PL/SQL blocks.
- 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.

❑ The PL/SQL Identifiers

- PL/SQL identifiers are constants, variables, exceptions, procedures, cursors, and reserved words.

- The identifiers consist of a letter optionally followed by more letters, numerals, dollar signs, underscores, and number signs and should not exceed 30 characters.

- By default, **identifiers are not case-sensitive**. So you can use **integer** or **INTEGER** to represent a numeric value. You cannot use a reserved keyword as an identifier.

❑ The PL/SQL Comments

- Program comments are explanatory statements that can be included in the PL/SQL code that you write and helps anyone reading its source code.

- All programming languages allow some form of comments.

- The PL/SQL supports single-line and multi-line comments.

- All characters available inside any comment are ignored by the PL/SQL compiler.

- The PL/SQL single-line comments start with the delimiter -- (double hyphen) and multi-line comments are enclosed by /* and */.

```
DECLARE -- variable declaration .
      message varchar2(20):= 'Hello, World!';
 BEGIN
       /*  PL/SQL executable statement(s) */
dbms_output.put_line(message);
 END;
/
```

- **Output:-**

```
Hello World
PL/SQL procedure successfully completed.
```

PL/SQL - Data Types

- CHARACTER Data type
- NUMBER Data type
- BOOLEAN Data type
- DATE Data type
- LOB Data type

PL/SQL - Data Types

S.No	Category & Description
1	**Scalar** Single values with no internal components, such as a **NUMBER, DATE,** or **BOOLEAN**.
2	**Large Object (LOB)** Pointers to large objects that are stored separately from other data items, such as text, graphic images, video clips, and sound waveforms.
3	**Composite** Data items that have internal components that can be accessed individually. For example, collections and records.
4	**Reference** Pointers to other data items.

PL/SQL - Variables

- The name of a PL/SQL variable consists of a letter optionally followed by more letters, numerals, dollar signs, underscores, and number signs and should not exceed 30 characters.

- By default, variable names are not case-sensitive. You cannot use a reserved PL/SQL keyword as a variable name.

- PL/SQL programming language allows to define various types of variables, such as date time data types, records, collections, etc

Variable Declaration in PL/SQL

- PL/SQL variables must be declared in the declaration section or in a package as a global variable. When you declare a variable, PL/SQL allocates memory for the variable's value and the storage location is identified by the variable name.

- **The syntax for declaring a variable is –**

  variable_name [CONSTANT] datatype [NOT NULL] [:= | DEFAULT initial_value]

- Where, *variable_name* is a valid identifier in PL/SQL, *datatype* must be a valid PL/SQL data type.

- Some valid variable declarations along with their definition are shown below −

```
sales number(10, 2);
pi CONSTANT double precision := 3.1415;
 name varchar2(25);
address varchar2(100);
```

- When you provide a size, scale or precision limit with the data type, it is called a **constrained declaration**.

- Constrained declarations require less memory than unconstrained declarations.

- **For example –**

```
sales number(10, 2);
name varchar2(25);
address varchar2(100);
```

Initializing Variables in PL/SQL

- Whenever you declare a variable, PL/SQL assigns it a default value of NULL.

- If you want to initialize a variable with a value other than the NULL value, you can do so during the declaration, using either of the following –

- The **DEFAULT** keyword
- The **assignment** operator

- **For example –**

```
counter integer := 0;
greetings varchar2(20) DEFAULT 'Have a Good Day';
```

- You can also specify that a variable should not have a **NULL** value using the **NOT NULL** constraint.
- If you use the NOT NULL constraint, you must explicitly assign an initial value for that variable.
- It is a good programming practice to initialize variables properly otherwise, sometimes programs would produce unexpected results.

```
DECLARE
    a integer := 10;
    b integer := 20;
    c integer;
    f real;
BEGIN
    c := a + b;
    dbms_output.put_line('Value of c: ' || c);
    f := 70.0/3.0;
    dbms_output.put_line('Value of f: ' || f);
 END;
/
```

- **Output**

```
Value of c: 30
Value of f: 23.433333333333333333

PL/SQL procedure successfully completed.
```

Variable Scope in PL/SQL

- PL/SQL allows the nesting of blocks, i.e., each program block may contain another inner block. If a variable is declared within an inner block, it is not accessible to the outer block.

- However, if a variable is declared and accessible to an outer block, it is also accessible to all nested inner blocks.

- **There are two types of variable scope –**
- **Local variables** − Variables declared in an inner block and not accessible to outer blocks.
- **Global variables** − Variables declared in the outermost block or a package.

```
DECLARE
    -- Global variables
    num1 number := 95;
    num2 number := 85;
BEGIN
     dbms_output.put_line('Outer Variable num1: ' || num1);
     dbms_output.put_line('Outer Variable num2: ' || num2);
     DECLARE
          -- Local variables
          num1 number := 195;
         num2 number := 185;
         BEGIN
         dbms_output.put_line('Inner Variable num1: ' || num1);
         dbms_output.put_line('Inner Variable num2: ' || num2);
         END;
END;
 /
```

- **Output**

```
Outer Variable num1: 95
Outer Variable num2: 85
Inner Variable num1: 195
Inner Variable num2: 185
```

- PL/SQL procedure successfully completed.

PL/SQL Records

❖ What are records?

- Records are another type of datatypes which oracle allows to be defined as a placeholder.
- Records are composite datatypes, which means it is a combination of different scalar datatypes like char, varchar, number etc.
- Each scalar data types in the record holds a value. A record can be visualized as a row of data. It can contain all the contents of a row.

- A **record** is a data structure that can hold data items of different kinds. Records consist of different fields, similar to a row of a database table.

- For example, you want to keep track of your books in a library. You might want to track the following attributes about each book, such as **Title, Author, Subject, Book ID.**

- A record containing a field for each of these items allows treating a BOOK as a logical unit and allows you to organize and represent its information in a better way.

- **PL/SQL can handle the following types of records –**

✔ Table-based
✔ Cursor-based records
✔ User-defined records

❖ Table-Based Records

- The %ROWTYPE attribute enables a programmer to create **table-based** and **cursorbased** records.

- The following example illustrates the concept of **table-based** records.

- Consider CUSTOMERS table,

```
DECLARE
     customer_rec customers%rowtype;

BEGIN
     SELECT * into customer_rec
      FROM      customers WHERE id = 5;

     dbms_output.put_line('Customer ID: ' || customer_rec.id);

     dbms_output.put_line('Customer Name: ' || customer_rec.name);

     dbms_output.put_line('Customer Address: ' ||customer_rec.address);

     dbms_output.put_line('Customer Salary: ' || customer_rec.salary);

END;
/
```

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

```
Customer ID: 5
Customer Name: Hardik
Customer Address: Bhopal
 Customer Salary: 9000
PL/SQL procedure successfully completed.
```

❖ Cursor-Based Records

- The following example illustrates the concept of **cursor-based** records.

```
DECLARE
    CURSOR customer_cur is
     SELECT id, name, address
     FROM customers;
          customer_rec customer_cur%rowtype;
BEGIN
     OPEN customer_cur;
     LOOP
          FETCH customer_cur into customer_rec;
           EXIT WHEN customer_cur%notfound;
          DBMS_OUTPUT.put_line(customer_rec.id || ' ' ||
customer_rec.name);
     END LOOP;
END; /
```

- **Output**

```
1 Ramesh
2 Khilan
3 kaushik
4 Chaitali
5 Hardik
6 Komal
```

- PL/SQL procedure successfully completed.

❖ Assigning SQL Query Results to PL/SQL Variables

- You can use the **SELECT INTO** statement of SQL to assign values to PL/SQL variables. For each item in the **SELECT list**, there must be a corresponding, type-compatible variable in the **INTO list**.

- Consider following example, to create a table named **CUSTOMERS**

```
CREATE TABLE CUSTOMERS(
                    ID INT NOT NULL,
                    NAME VARCHAR (20) NOT NULL,
                    AGE INT NOT NULL,
                   ADDRESS CHAR (25),
                    SALARY DECIMAL (18, 2),
                   PRIMARY KEY (ID) );
```

- Table Created.

- **Insert some values in the table –**
- INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY) VALUES (1, 'Ramesh', 32, 'Ahmedabad', 2000.00);
- INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY) VALUES (2, 'Khilan', 25, 'Delhi', 1500.00);
- INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY) VALUES (3, 'kaushik', 23, 'Kota', 2000.00);
- INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY) VALUES (4, 'Chaitali', 25, 'Mumbai', 6500.00);
- INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY) VALUES (5, 'Hardik', 27, 'Bhopal', 8500.00);
- INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY) VALUES (6, 'Komal', 22, 'MP', 4500.00);

- The following program assigns values from the above table to PL/SQL variables using the **SELECT INTO clause** of SQL −

```
DECLARE
      c_id customers.id%type := 1;
      c_name customers.name%type;
      c_addr customers.address%type;
      c_sal customers.salary%type;
BEGIN
           SELECT name, address, salary INTO c_name, c_addr, c_sal
FROM customers
      WHERE id = c_id;
      dbms_output.put_line ('Customer ' ||c_name || ' from ' || c_addr || ' earns ' ||
c_sal);
END;
/
```

- **When the above code is executed, it produces the following result −**

```
Customer Ramesh from Ahmedabad earns 2000

 PL/SQL procedure completed successfully
```

CONTROL STRUCTURES: CONDITIONAL CONTROLS

Conditional Control

- Conditional control allows you to control the flow of the execution of the program based on a condition.
- In programming terms, it means that the statements in the program are not executed sequentially.
- Rather, one group of statements, or another will be executed depending on how the condition is evaluated.
- In PL/SQL, there are two types of conditional control:
  - IF statement and
  - ELSIF statement.

IF STATEMENTS

- An IF statement has two forms:
  IF-THEN and IF-THEN-ELSE.

- An IF-THEN statement allows you to specify only one group of actions to take.

- In other words, this group of actions is taken only when a condition evaluates to TRUE.

- And IF-THEN-ELSE statement allows you to specify two groups of actions, and the second group of actions is taken when a condition evaluates to FALSE.

IF-THEN STATEMENTS

- An IF-THEN statement is the most basic kind of a conditional control and has the following structure:

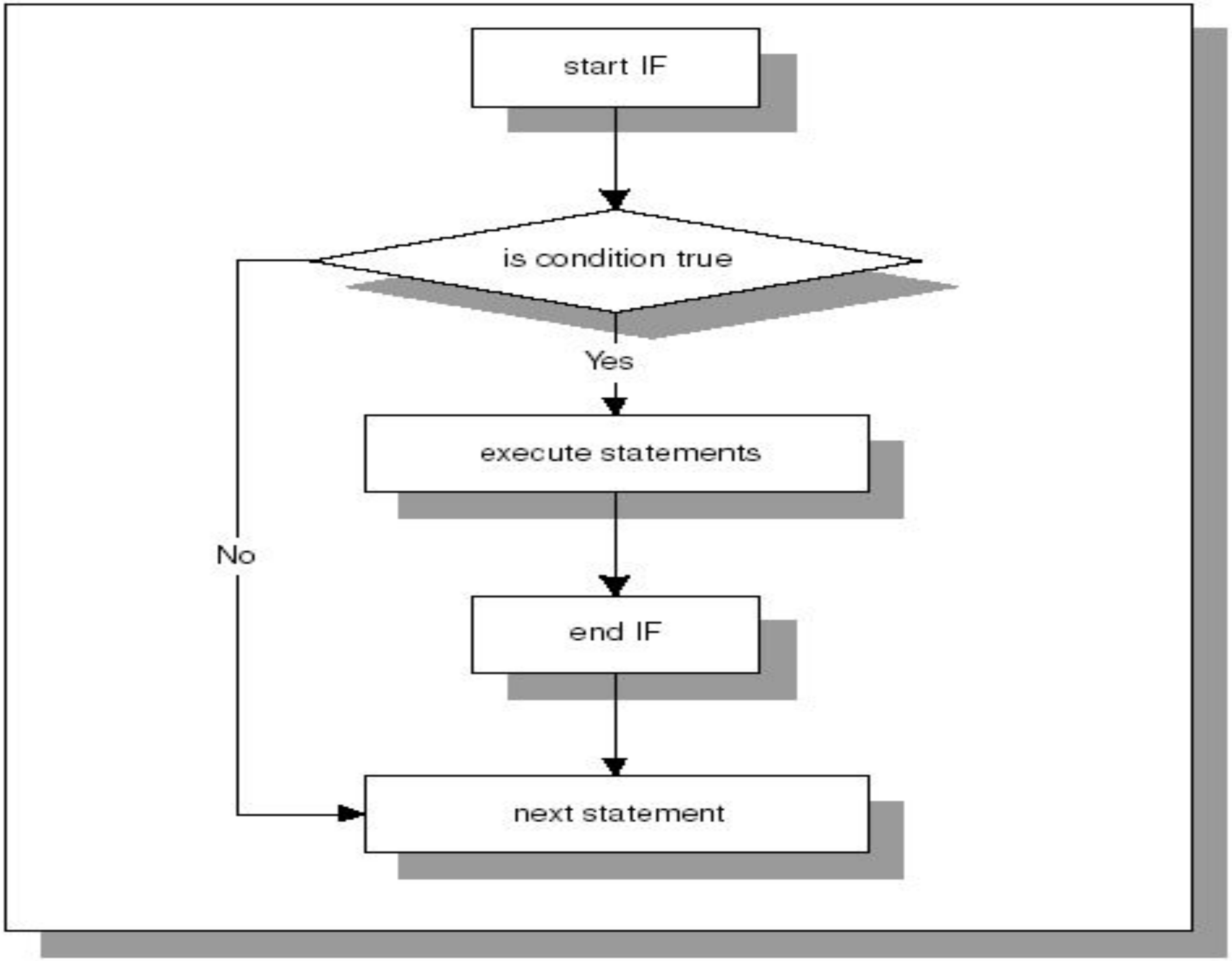
```
IF CONDITION
THEN
    STATEMENT 1;
    …
    STATEMENT N;
END IF;
```

- The reserved word IF marks the beginning of the IF statement.
- Statements 1 through N are a sequence of executable statements that consist of one or more of the standard programming structures.

IF-THEN STATEMENTS

- The word *CONDITION* between keywords IF and THEN determines whether these statements are executed.

- END IF is a reserved phrase that indicates the end of the IF-THEN construct.


start IF
is condition true
Yes
No
execute statements
end IF
next statement

PL/SQL - Basic Loop Statement

- Basic loop structure encloses sequence of statements in between the **LOOP** and **END LOOP** statements. With each iteration, the sequence of statements is executed and then control resumes at the top of the loop.

- **Syntax**
- The syntax of a basic loop in PL/SQL programming language is −

```
LOOP
    Sequence of statements;
END LOOP;
```

- Here, the sequence of statement(s) may be a single statement or a block of statements.
- An **EXIT statement** or an **EXIT WHEN statement** is required to break the loop.

- **Example**

```
DECLARE
      x number := 10;
BEGIN
     LOOP
      dbms_output.put_line(x);
      x := x + 10;
            IF x > 50 THEN
      exit;
            END IF;
     END LOOP;
 -- after exit, control resumes here
dbms_output.put_line('After Exit x is: ' || x);
 END;
 /
```

- **Output**

```
                 10 20 30 40 50
                 After Exit x is: 60
```

- PL/SQL procedure successfully completed.

- You can use the **EXIT WHEN** statement instead of the **EXIT** statement −

```
DECLARE
    x number := 10;
BEGIN
    LOOP
        dbms_output.put_line(x);
        x := x + 10;
        exit WHEN x > 50;
     END LOOP;
-- after exit, control resumes here
    dbms_output.put_line('After Exit x is: ' || x);
END;
 /
```

- **Output**

```
10
20
30
40
50
After Exit x is: 60
PL/SQL procedure successfully completed.
```

- Loops concept provides the following advantage in coding
- Reusability of code
- Reduced code size
- Easy flow of control
- Reduced Complexity

Loop Control Statements

Loop control statements are those that actually controls the flow of execution inside the loop. Below is the detailed description about the loop control statements.

❖ CONTINUE

- This keyword sends an instruction to the PL/SQL engine that whenever PL/SQL engine encounters this keyword inside the loop, then it will skip the remaining code in the execution block of the code, and next iteration will start immediately. This will be mainly used if the code inside the loop wants to be skipped for certain iteration values.

EXIT / EXIT WHEN

- This keyword sends an instruction to the PL/SQL engine that whenever PL/SQL engine encounters this keyword, then it will immediately exit from the current loop.

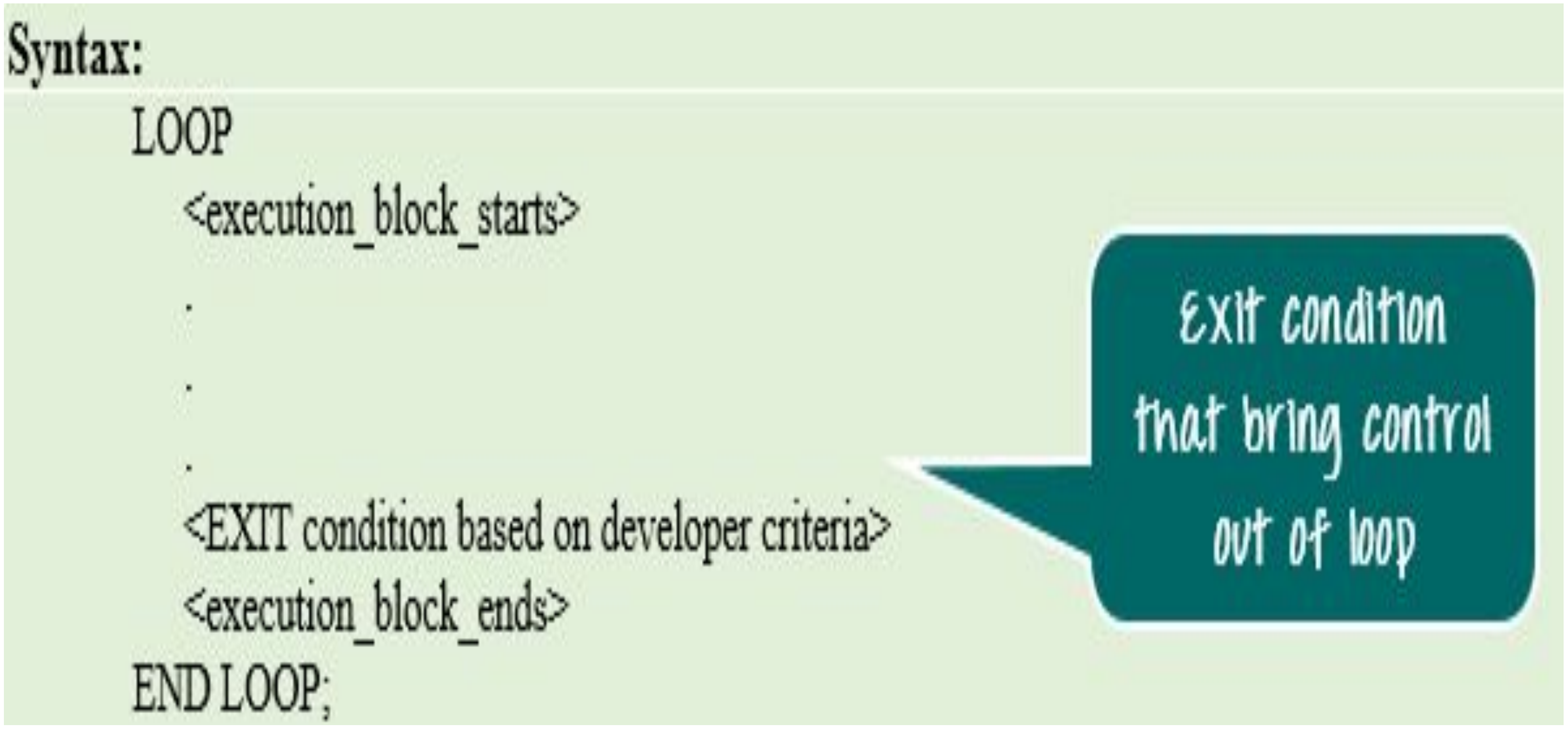
- If the PL/SQL engine encounters the EXIT in nested loops, then it will come out of the loop in which it has been defined, i.e. in a nested loops, giving EXIT in the inner loop will only exit the control from inner loop but not from the outer loop. 'EXIT WHEN' is followed by an expression which gives Boolean result.

- If the result is TRUE, then the control will EXIT.

GOTO

- This statement will transfer the control to the labeled statement ("GOTO <label> ;").
- Transfer of control can be done only within the subprograms.
- Transfer of control cannot be done from exception handling part to the execution part
- Usage of this statement is not recommended unless there are no other alternatives, as the code-control traceability will be very difficult in the program due to the transfer of control from one part to another part.

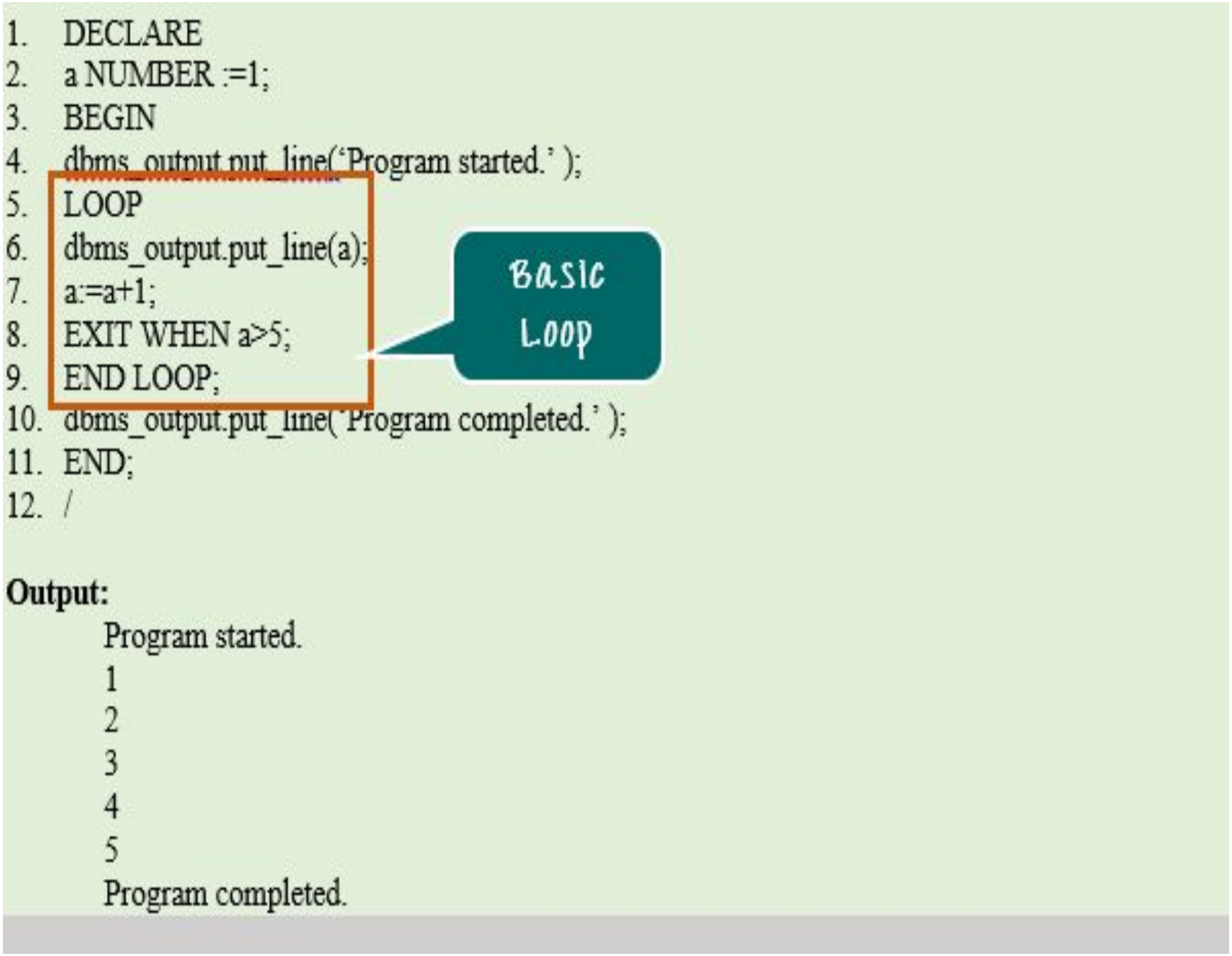
Basic Loop Statement



Syntax:

```
    LOOP
      <execution_block_starts>
      .
      .
      .
      <EXIT condition based on developer criteria>
      <execution_block_ends>
    END LOOP;
```

Exit condition that bring control out of loop



```
DECLARE
a NUMBER :=1;
BEGIN
dbms_output.put_line('Program started.' );
LOOP
dbms_output.put_line(a);
a:=a+1;
EXIT WHEN a>5;
END LOOP;
dbms_output.put_line('Program completed.' );
END;
/
```

Output:

```
Program started.
1
2
3
4
5
Program completed.
```

PL/SQL - FOR LOOP Statement

- A **FOR LOOP** is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.
- **Syntax**

```
FOR counter IN initial_value .. final_value
 LOOP
sequence_of_statements;
END LOOP;
```

- **Following is the flow of control in a For Loop −**
- The initial step is executed first, and only once. This step allows you to declare and initialize any loop control variables.

- Next, the condition, i.e., *initial_value .. final_value* is evaluated. If it is TRUE, the body of the loop is executed. If it is FALSE, the body of the loop does not execute and the flow of control jumps to the next statement just after the for loop.

- After the body of the for loop executes, the value of the counter variable is increased or decreased.

- The condition is now evaluated again. If it is TRUE, the loop executes and the process repeats itself (body of loop, then increment step, and then again condition). After the condition becomes FALSE, the FOR-LOOP terminates.

- **Following are some special characteristics of PL/SQL for loop**
- The *initial_value* and *final_value* of the loop variable or counter can be literals, variables, or expressions but must evaluate to numbers. Otherwise, PL/SQL raises the predefined exception VALUE_ERROR.

- The *initial_value* need not be 1; however, the **loop counter increment (or decrement) must be 1**.

- PL/SQL allows the determination of the loop range dynamically at run time.

- **Example**

```
DECLARE
      a number(2);
 BEGIN
 FOR a in 10 .. 20
 LOOP
dbms_output.put_line('value of a: ' || a);
END LOOP;
END;
 /
```

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

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
 value of a: 14
value of a: 15
value of a: 16
value of a: 17
 value of a: 18
value of a: 19
value of a: 20

PL/SQL procedure successfully completed.
```

Reverse FOR LOOP Statement

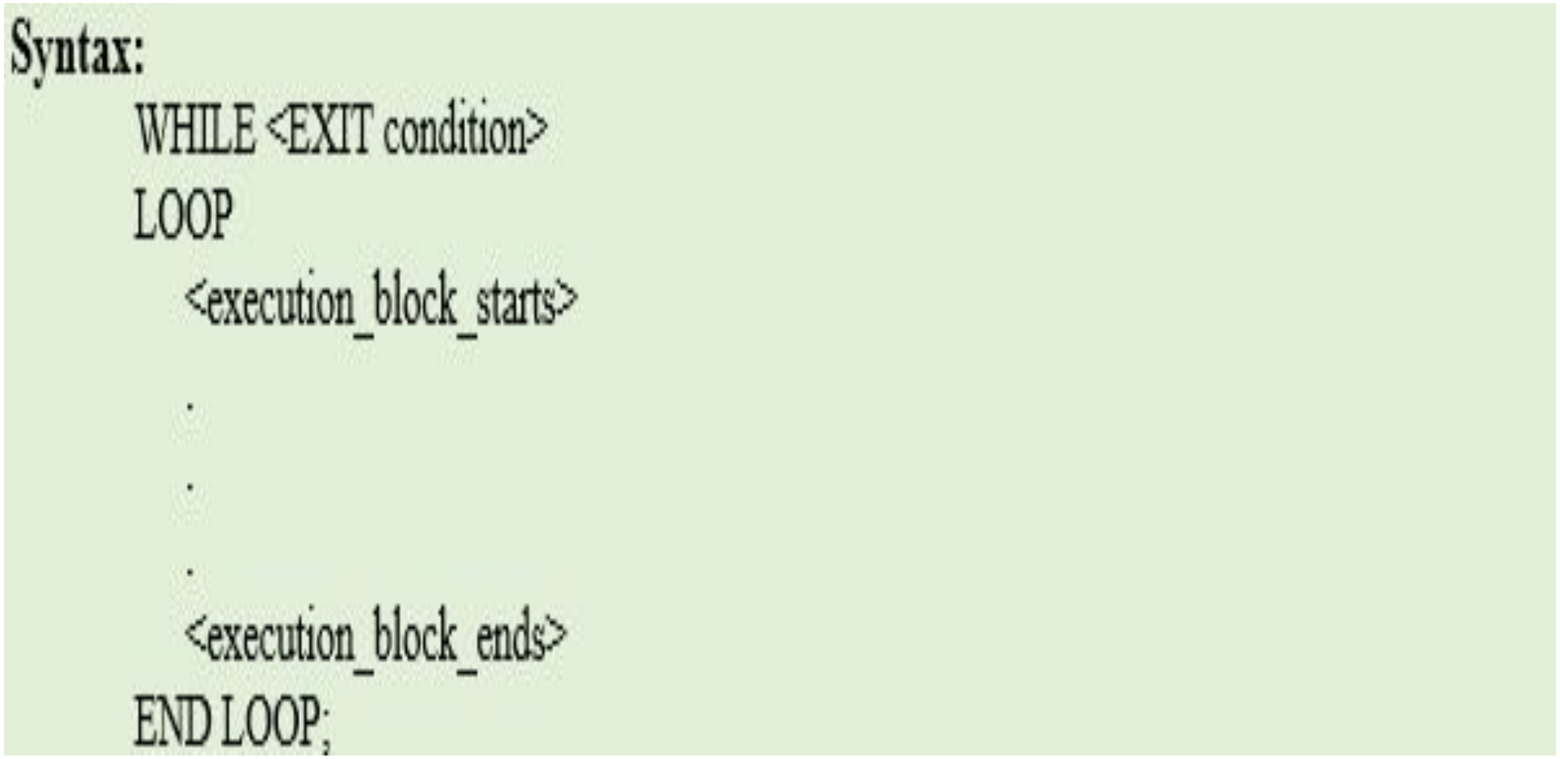
- By default, iteration proceeds from the initial value to the final value, generally upward from the lower bound to the higher bound. You can reverse this order by using the **REVERSE** keyword.
- In such case, iteration proceeds the other way. After each iteration, the loop counter is decremented.
- However, you must write the range bounds in ascending (not descending) order.

```
DECLARE a number(2) ;
BEGIN
FOR a IN REVERSE 10 .. 20
 LOOP
 dbms_output.put_line('value of a: ' || a);
END LOOP;
END;
/
```

- value of a: 20
- value of a: 19
- value of a: 18
- value of a: 17
- value of a: 16
- value of a: 15
- value of a: 14
- value of a: 13
- value of a: 12
- value of a: 11
- value of a: 10

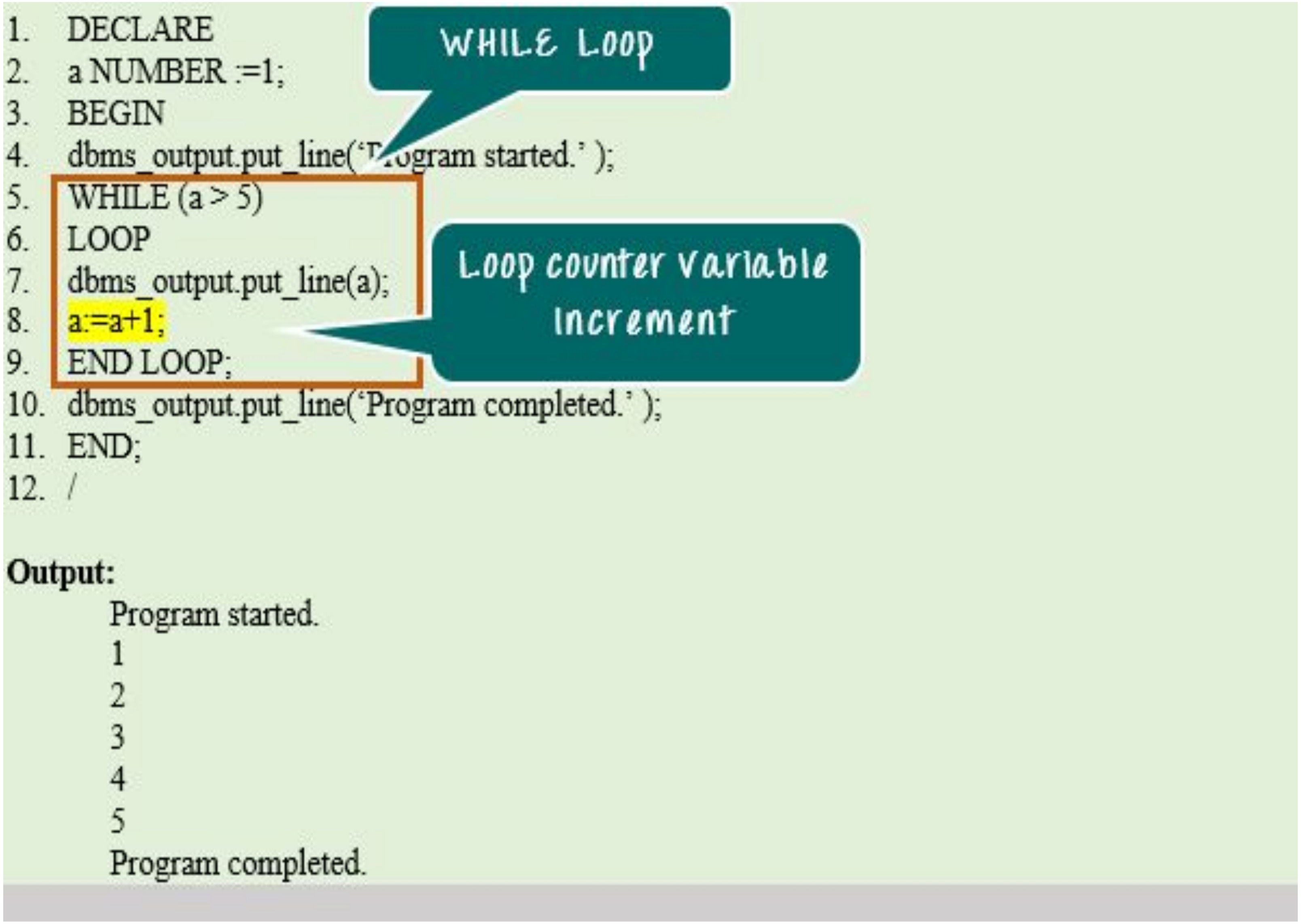
- PL/SQL procedure successfully completed.

While Loop



Syntax:

```
    WHILE <EXIT condition>
    LOOP
        <execution_block_starts>
        .
        .
        .
        <execution_block_ends>
    END LOOP;
```


WHILE Loop
Loop counter variable increment

```
DECLARE
a NUMBER :=1;
BEGIN
dbms_output.put_line('Program started.' );
WHILE (a > 5)
LOOP
dbms_output.put_line(a);
a:=a+1;
END LOOP;
dbms_output.put_line('Program completed.' );
END;
/
```

Output:

```
Program started.
1
2
3
4
5
Program completed.
```

PL/SQL - Exceptions

- An exception is an error condition during a program execution.

- PL/SQL supports programmers to catch such conditions using **EXCEPTION** block in the program and an appropriate action is taken against the error condition.

- **There are two types of exceptions –**

✔ System-defined exceptions

✔ User-defined exceptions

- **Syntax for Exception Handling**
- The general syntax for exception handling is as follows. Here you can list down as many exceptions as you can handle.
- The default exception will be handled using ***WHEN others THEN*** –

```
DECLARE
    <declarations section>
BEGIN
    <executable command(s)>
EXCEPTION
    <exception handling goes here >
    WHEN    exception1 THEN
        exception1-handling-statements
    WHEN   exception2 THEN
        exception2-handling-statements
    WHEN   exception3 THEN
         exception3-handling-statements ........
    WHEN   others THEN
        exception3-handling-statements
END;
```

```
DECLARE
 c_id customers.id%type := 8;
 c_name customers.name%type;
c_addr customers.address%type;
BEGIN
	SELECT name, address INTO c_name, c_addr
	FROM customers WHERE id = c_id;
	DBMS_OUTPUT.PUT_LINE ('Name: '|| c_name);
	DBMS_OUTPUT.PUT_LINE ('Address: ' || c_addr);
EXCEPTION WHEN no_data_found THEN
	dbms_output.put_line('No such customer!');
	WHEN others THEN dbms_output.put_line('Error!');
END;
 /
```

- No such customer!
- PL/SQL procedure successfully completed.

Raising Exceptions

- Exceptions are raised by the database server automatically whenever there is any internal database error, but exceptions can be raised explicitly by the programmer by using the command **RAISE**.

- **Syntax for raising an exception −**

```
DECLARE
    exception_name EXCEPTION;
BEGIN
    IF condition THEN
        RAISE exception_name;
    END IF;
 EXCEPTION
    WHEN exception_name
THEN
    statement;
END;
```

User-defined Exceptions

- PL/SQL allows you to define your own exceptions according to the need of your program.
- A user-defined exception must be declared and then raised explicitly, using either a RAISE statement or the procedure **DBMS_STANDARD.RAISE_APPLICATION_ERROR**.

- The syntax for declaring an exception is –

```
DECLARE my-exception EXCEPTION;
```

- **Example**
- The program asks for a customer ID, when the user enters an invalid ID, the exception **invalid_id** is raised.

```
DECLARE
c_id customers.id%type := &cc_id;
c_name customers.name%type;
c_addr customers.address%type; -- user
defined exception
 ex_invalid_id EXCEPTION;
 BEGIN
      IF c_id <= 0 THEN
      RAISE ex_invalid_id;
      ELSE
      SELECT name, address INTO
c_name, c_addr FROM customers
      WHERE id = c_id;
      DBMS_OUTPUT.PUT_LINE
('Name: '|| c_name);
DBMS_OUTPUT.PUT_LINE ('Address:
' || c_addr);
     END IF;
EXCEPTION
  WHEN ex_invalid_id  THEN
dbms_output.put_line('ID    must be
greater than zero!');
  WHEN no_data_found THEN
dbms_output.put_line('No    such
customer!');
  WHEN others THEN
dbms_output.put_line('Error!'); END;
/
```

```
Enter value for cc_id: -6 (let's enter a value -6)
old 2: c_id customers.id%type := &cc_id;
new 2: c_id customers.id%type := -6;

ID must be greater than zero!

PL/SQL procedure successfully completed.
```

❖ Pre-defined Exceptions

- PL/SQL provides many pre-defined exceptions, which are executed when any database rule is violated by a program.

- **For example**, the predefined exception NO_DATA_FOUND is raised when a SELECT INTO statement returns no rows.

- The following table lists few of the important pre-defined exceptions −

Exception	Oracle Error	SQLCODE	Description
ACCESS_INTO_NULL	06530	-6530	It is raised when a null object is automatically assigned a value.
CASE_NOT_FOUND	06592	-6592	It is raised when none of the choices in the WHEN clause of a CASE statement is selected, and there is no ELSE clause.
COLLECTION_IS_NULL	06531	-6531	It is raised when a program attempts to apply collection methods other than EXISTS to an uninitialized nested table or varray, or the program attempts to assign values to the elements of an uninitialized nested table or varray.
DUP_VAL_ON_INDEX	00001	-1	It is raised when duplicate values are attempted to be stored in a column with unique index.

INVALID_CURSOR	01001	-1001	It is raised when attempts are made to make a cursor operation that is not allowed, such as closing an unopened cursor.
INVALID_NUMBER	01722	-1722	It is raised when the conversion of a character string into a number fails because the string does not represent a valid number.
LOGIN_DENIED	01017	-1017	It is raised when a program attempts to log on to the database with an invalid username or password.
NO_DATA_FOUND	01403	+100	It is raised when a SELECT INTO statement returns no rows.

NOT_LOGGED_ON	**01012**	**-1012**	**It is raised when a database call is issued without being connected to the database.**
PROGRAM_ERROR	06501	-6501	It is raised when PL/SQL has an internal problem.
ROWTYPE_MISMATCH	06504	-6504	It is raised when a cursor fetches value in a variable having incompatible data type.
SELF_IS_NULL	30625	-30625	It is raised when a member method is invoked, but the instance of the object type was not initialized.

STORAGE_ERROR	06500	-6500	It is raised when PL/SQL ran out of memory or memory was corrupted.
TOO_MANY_ROWS	01422	-1422	It is raised when a SELECT INTO statement returns more than one row.
VALUE_ERROR	06502	-6502	It is raised when an arithmetic, conversion, truncation, or sizeconstraint error occurs.
ZERO_DIVIDE	01476	1476	It is raised when an attempt is made to divide a number by zero.

Stored Procedures & Functions

Stored Procedures & Functions

- A **subprogram** is a program unit/module that performs a particular task. These subprograms are combined to form larger programs. This is basically called the 'Modular design'.
- A subprogram can be invoked by another subprogram or program which is called the **calling program**.

A subprogram can be created –

- ✓ At the schema level
- ✓ Inside a package
- ✓ Inside a PL/SQL block

- At the schema level, subprogram is a **standalone subprogram**.
- It is created with the **CREATE PROCEDURE or the CREATE FUNCTION** statement.
- It is stored in the database and can be deleted with the **DROP PROCEDURE or DROP FUNCTION** statement.
- A subprogram created inside a package is a **packaged subprogram**.
- It is stored in the database and can be deleted only when the package is deleted with the **DROP PACKAGE** statement.

What is a Stored Procedure?

- A **stored procedure** or in simple a **proc** is a named PL/SQL block which performs one or more specific task.

- This is similar to a procedure in other programming languages.

- A procedure has a header and a body. The header consists of the name of the procedure and the parameters or variables passed to the procedure.

- The body consists of declaration section, execution section and exception section similar to a general PL/SQL Block.

- PL/SQL subprograms are named PL/SQL blocks that can be invoked with a set of parameters.

- PL/SQL provides two kinds of subprograms −

- **Functions** − These subprograms return a single value; mainly used to compute and return a value.

- **Procedures** − These subprograms do not return a value directly; mainly used to perform an action.

❖ Creating a Procedure

- A procedure is created with the **CREATE OR REPLACE PROCEDURE** statement.

- **Syntax:-**
- CREATE [OR REPLACE] PROCEDURE procedure_name

```
 [(parameter_name [IN | OUT | IN OUT] type [, ...])]
{IS | AS}
BEGIN
    < procedure_body >
END procedure_name;
```

- Where,
- *procedure-name* specifies the name of the procedure.
- [OR REPLACE] option allows the modification of an existing procedure.
- 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.
- *procedure-body* contains the executable part.
- **IS** keyword is used, when the procedure is nested into some other blocks.
- The **AS** keyword is used instead of the **IS** keyword for creating a standalone procedure.

- **Example**
- The following example creates a simple procedure that displays the string 'Hello World!' on the screen when executed.

```
CREATE OR REPLACE PROCEDURE greetings
AS
BEGIN
   dbms_output.put_line('Hello World!');
END;
```

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

- Procedure created.

Executing a Standalone Procedure

- A standalone procedure can be called in two ways −
- Using the **EXECUTE** keyword
- Calling the name of the procedure from a PL/SQL block
- The above procedure named **'greetings'** can be called with the EXECUTE keyword as −

```
EXECUTE greetings;
```

- The above call will display −

```
Hello World
        PL/SQL procedure successfully completed.
```

- **The procedure can also be called from another PL/SQL block −**

```
BEGIN
 greetings;
END;
/
```

- The above call will display −

```
Hello World
PL/SQL procedure successfully completed.
```

- **Deleting a Standalone Procedure**
- A standalone procedure is deleted with the **DROP PROCEDURE** statement.

- Syntax for deleting a procedure is –

```
DROP PROCEDURE procedure-name;
```

- You can drop the greetings procedure by using the following statement –

```
DROP PROCEDURE greetings;
```

- **Procedures: Passing Parameters**
- We can pass parameters to procedures in three ways.
  1) IN-parameters
  2) OUT-parameters
  3) IN OUT-parameters

- A procedure may or may not return any value.

✔ **IN**

An IN parameter lets you pass a value to the subprogram. **It is a read-only parameter**. Inside the subprogram, an IN parameter acts like a constant. It cannot be assigned a value. You can pass a constant, initialized variable, or expression as an IN parameter. You can also initialize it to a default value; however, in that case, it is omitted from the subprogram call. **It is the default mode of parameter passing. Parameters are passed by reference**.

✔ **OUT**

An OUT parameter returns a value to the calling program. Used for getting output from the subprograms. Inside the subprogram, an OUT parameter acts like a variable. You can change its value and reference the value after assigning it. **The actual parameter must be variable and it is passed by value**.

✔ IN OUT

- An **IN OUT** parameter passes an initial value to a subprogram and returns an updated value to the caller. It can be assigned a value and the value can be read.

- The actual parameter corresponding to an IN OUT formal parameter must be a variable, not a constant or an expression.

- Formal parameter must be assigned a value. **Actual parameter is passed by value.**

```
DECLARE
a number;
 b number;
c number;
PROCEDURE findMin(x IN number, y IN number, z OUT number) IS
BEGIN
      IF x < y THEN
             z:= x;
      ELSE z:= y;
      END IF;
END;
BEGIN
      a:= 23;
       b:= 45;
      findMin(a, b, c);
       dbms_output.put_line(' Minimum of (23, 45) : ' || c);
END; /
```

- **Output:-**

```
Minimum of (23, 45) : 23
PL/SQL procedure successfully completed.
```

- **Example:-**
- This procedure computes the square of value of a passed value. This example shows how we can use the same parameter to accept a value and then return another result.

```
DECLARE
    a number;
PROCEDURE squareNum(x IN OUT number) IS
BEGIN
    x := x * x;
END;
BEGIN
    a:= 23;
    squareNum(a);
    dbms_output.put_line(' Square of (23): ' || a);
END;
/
```

- **Output**

```
Square of (23): 529
PL/SQL procedure successfully completed.
```

❖ Methods for Passing Parameters

- Actual parameters can be passed in three ways –
  - Positional notation:-findMin(a, b, c, d);
  - Named notation:-findMin(x => a, y => b, z => c, m => d);
  - Mixed notation:-findMin(a, b, c, m => d);

FUNCTIONS

FUNCTIONS

- Functions are a type of stored code and are very similar to procedures.
- The significant difference is that a function is a PL/SQL block that *returns* a single value.
- Functions can accept one, many, or no parameters, but a function must have a return clause in the executable section of the function.
- The datatype of the return value must be declared in the header of the function.
- A function has output that needs to be assigned to a variable, or it can be used in a SELECT statement.

- The function does not necessarily have to have any parameters, but it must have a RETURN value declared in the header, and it must return values for all the varying possible execution streams.

- The RETURN statement does not have to appear as the last line of the main execution section, and there may be more than one RETURN statement (there should be a RETURN statement for each exception).

- A function may have IN, OUT, or IN OUT parameters.

FUNCTIONS

- A function is a named PL/SQL Block which is similar to a procedure. The major difference between a procedure and a function is, a function must always return a value, but a procedure may or may not return a value.
- **General Syntax to create a function is**

```
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

```
CREATE OR REPLACE FUNCTION totalCustomers
RETURN number
IS total number(2) := 0;
BEGIN
   SELECT count(*) into total FROM customers;
RETURN total;
 END;
/
```

- **Output:-**

```
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;
/
```

- **Output:-**
- Total no. of Customers: 6
- PL/SQL procedure successfully completed.

- **Example:-**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;
/
```

Output:-

```
Maximum of (23,45): 45
PL/SQL procedure successfully
completed.
```

References

- https://www.tutorialspoint.com/plsql/
- https://en.wikipedia.org/wiki/PL/SQL

END