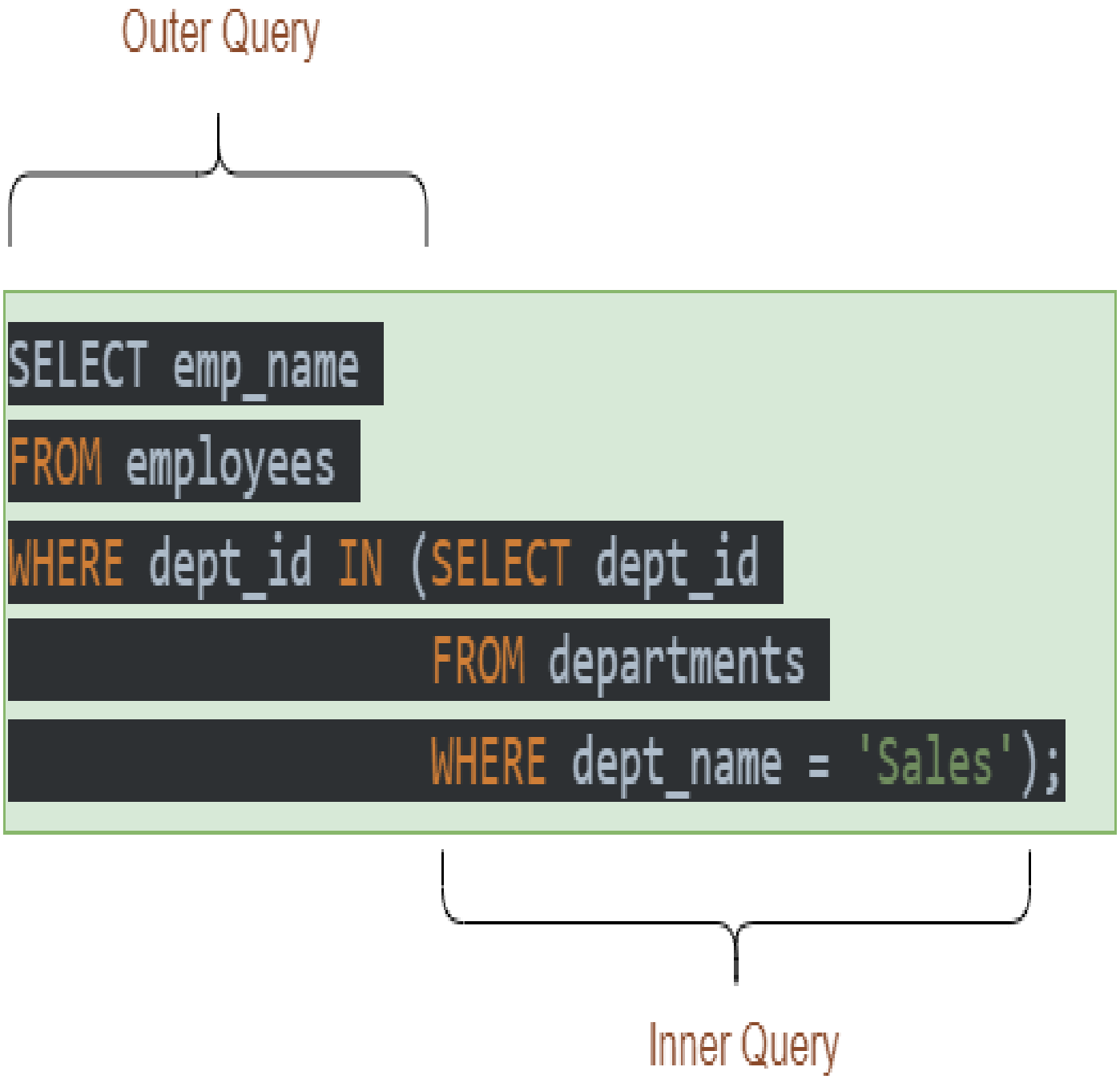
Nested Queries

- In SQL, a nested query involves a query that is placed within another query. Output of the inner query is used by the outer query. A nested query has two SELECT statements: one for the inner query and another for the outer query.


Outer Query
SELECT emp_name
FROM employees
WHERE dept_id IN (SELECT dept_id
FROM departments
WHERE dept_name = 'Sales');
Inner Query

- SELECT column1, column2, ... FROM table1 WHERE column1 IN (SELECT column1 FROM table2 WHERE condition);

Operators

IN Operator

- This operator checks if a column value in the outer query's result is present in the inner query's result. The final result will have rows that satisfy the IN condition.

NOT IN Operator

- This operator checks if a column value in the outer query's result is not present in the inner query's result. The final result will have rows that satisfy the NOT IN condition.

ALL Operator

- This operator compares a value of the outer query's result with all the values of the inner query's result and returns the row if it matches all the values.

ANY Operator

- This operator compares a value of the outer query's result with all the inner query's result values and returns the row if there is a match with any value.

- EXISTS Operator

This operator checks whether a subquery returns any row. If it returns at least one row. EXISTS operator returns true, and the outer query continues to execute. If the subquery returns no row, the EXISTS operator returns false, and the outer query stops execution.

Table: employees table

emp_id	emp_name	dept_id
1	John	1
2	Mary	2
3	Bob	1
4	Alice	3
5	Tom	1

Table: departments table

dept_id	dept_name
1	Sales
2	Marketing
3	Finance

Table: sales table

sale_id	emp_id	sale_amt
1	1	1000
2	2	2000
3	3	3000
4	1	4000
5	5	5000
6	3	6000
7	2	7000

- Find the names of all employees in the sales department:

SELECT emp_name FROM employees WHERE dept_id IN (SELECT dept_id FROM departments WHERE dept_name = 'Sales');

emp_name
John
Bob
Tom

- Find the names of all employees who have made a sale:

SELECT emp_name FROM employees WHERE EXISTS (SELECT emp_id FROM sales WHERE employees.emp_id = sales.emp_id);

emp_name
John
Mary
Bob
Alice
Tom

- Find the names of all employees who have made sales greater than $1000.

SELECT emp_name FROM employees WHERE emp_id = ALL (SELECT emp_id FROM sales WHERE sale_amt > 1000);

emp_name
John
Mary
Bob
Tom

Aggregate Operators

- SQL aggregation function is used to perform the calculations on multiple rows of a single column of a table. It returns a single value.
- It is also used to summarize the data.

1. Count()

- The COUNT() function returns the number of rows that matches a specified criterion.
- SELECT COUNT(*column_name*)
  FROM *table_name*
  WHERE *condition*;

2. AVG()

- The AVG() function returns the average value of a numeric column.
- SELECT AVG(*column_name*)
  FROM *table_name*
  WHERE *condition*;

3. SUM()

- The SUM() function returns the total sum of a numeric column.
- SELECT SUM(*column_name*)
  FROM *table_name*
  WHERE *condition*;

4. Min() and Max()

- The MIN() function returns the smallest value of the selected column.
- The MAX() function returns the largest value of the selected column.
- SELECT MIN(column name)
  FROM table_name;
- SELECT MAX(column name)
  FROM table_name;

ProductID	ProductName	SupplierID	CategoryID	Quantity	Price
1	Chais	1	1	12	18
2	Chang	1	1	30	19
3	Aniseed Syrup	1	2	25	10
4	Chef Anton's Cajun Seasoning	2	2	10	22
5	Chef Anton's Gumbo Mix	2	2	3	21.35

1. No. of Records or products from the table:

- SELECT COUNT(ProductID)
  FROM Products;

2. Average price of all products

- SELECT AVG(Price)
  FROM Products;

3. Quantity of all products

- SELECT SUM(Quantity)
  FROM Products;

4. Lowest product price

- SELECT MIN(Price)
  FROM Products;

5. Highest Product price

- SELECT MAX(Price)
  FROM Products;

NULL VALUES

- A field with a NULL value is a field with no value.
- If a field in a table is optional, it is possible to insert a new record or update a record without adding a value to this field. Then, the field will be saved with a NULL value.

How to Test for NULL Values?

- It is not possible to test for NULL values with comparison operators, such as =, <, or <>.
- We will have to use the IS NULL and IS NOT NULL operators instead.

IS NULL

- SELECT *column_names*
  FROM *table_name*
  WHERE *column_name* IS NULL;

IS NOT NULL

- SELECT *column_names*
  FROM *table_name*
  WHERE *column_name* IS NOT NULL;

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico
4	Around the Horn	Thomas Hardy	120 Hanover Sq.	London	WA1 1DP	UK
						Germany

- Update Customers

Set CustomerId=5, Name=“Geetha”, ContactName=“Geetha”, Address=“Andhra”, City=“Tirupathi”, postalcode=048393

Where customerId IS NULL;

Complex Integrity Constraints

- In SQL, Integrity constraints are rules that ensure data integrity in a database.
- It can be classified into two : Simple and Complex
- Simple Integrity constraints include primary keys, unique, and foreign key that ensure data in a table is unique and consistent.

- On the other hand, complex integrity constraints are more advanced rules that cannot be expressed using simple constraints.
- They are typically used to enforce business rules, such as limiting the range of values that can be entered into a column or ensuring that certain combinations of values are present in a table.

- There are several ways to implement complex integrity constraints in sql.
  1. Using CHECK
  2. Using Composite Primary and foreign key
  3. Using Stored Procedure
  4. Using Triggers

1. Using CHECK Constraints

Ensuring a range: You might want a column to only have values within a certain range.

Example:

```
CREATE TABLE Employees (
        ID INT PRIMARY KEY,
        Age INT CHECK (Age >= 18 AND Age <= 30)
    );
```

Pattern matching: Ensure data in a column matches a particular format.

Example:

```
CREATE TABLE Students (
        ID INT PRIMARY KEY,
        Email VARCHAR(255) CHECK (Email LIKE '%@%.%')
    );
```

2. Composite Primary and Foreign Keys

These are cases where the uniqueness or referential integrity constraint is applied over more than one column.

Example:

```
CREATE TABLE OrderDetails (
        OrderID INT,
        ProductID INT,
        Quantity INT,
        PRIMARY KEY (OrderID, ProductID),
        FOREIGN KEY (OrderID) REFERENCES Orders(OrderID),
        FOREIGN KEY (ProductID) REFERENCES Products(ProductID)
    );
```

3. Stored Procedure

- Sometimes, instead of direct data manipulation on tables, using stored procedures can help maintain more complex integrity constraints by wrapping logic inside the procedure. For instance, you could have a procedure that checks several conditions before inserting a record.

4. Using Triggers

- A trigger is a stored procedural code in a database that automatically executes in response to certain events on a particular table or view. Essentially, triggers are special types of stored procedures that run automatically when an INSERT, UPDATE, or DELETE operation occurs.
- A trigger is a predefined action that the database automatically executes in response to certain events on a particular table or view. Triggers are typically used to maintain the integrity of the data, automate data-related tasks, and extend the database functionalities.

```
CREATE TABLE Order (
   ID INT PRIMARY KEY,
   CustomerID INT,
   OrderDate DATE,
   Amount DECIMAL(10,2),
   CONSTRAINT FK_Customer_Order FOREIGN KEY (CustomerID) REFERENCES Customer(ID)
);

CREATE TRIGGER TR_Order_Check_Amount
BEFORE INSERT ON Order
FOR EACH ROW
BEGIN
   IF NEW.Amount <= 0 THEN
      SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Amount must be greater than zero.';
   END IF;
END;
```

PL/SQL Introduction

PL/SQL Introduction

PL/SQL is a block structured language that enables developers to combine the power of SQL with procedural statements.All the statements of a block are passed to oracle engine all at once which increases processing speed and decreases the traffic.

Basics of PL/SQL

- •PL/SQL stands for Procedural Language extensions to the Structured Query Language (SQL).
- PL/SQL is a combination of SQL along with the procedural features of programming languages.
- Oracle uses a PL/SQL engine to processes the PL/SQL statements.
- PL/SQL includes procedural language elements like conditions and loops. It allows declaration of constants and variables, procedures and functions, types and variable of those types and triggers.

Disadvantages of SQL:

- SQL doesn't provide the programmers with a technique of condition checking, looping and branching.
- SQL statements are passed to Oracle engine one at a time which increases traffic and decreases speed.
- SQL has no facility of error checking during manipulation of data.

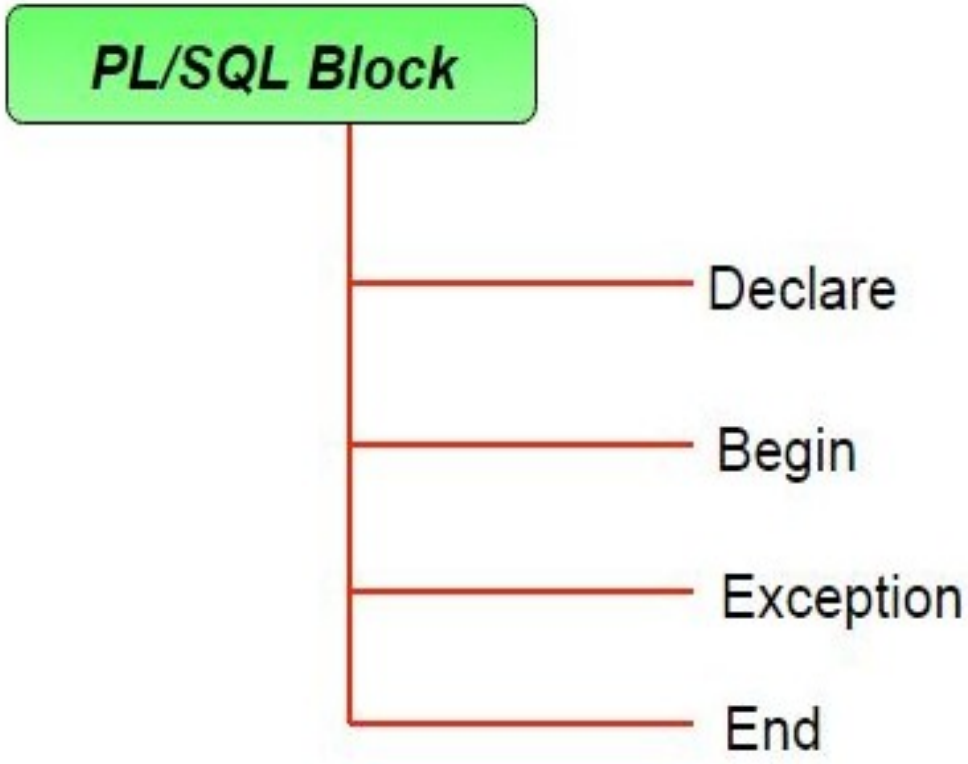
Features of PL/SQL:

1. PL/SQL is basically a procedural language, which provides the functionality of decision making, iteration and many more features of procedural programming languages.
2. PL/SQL can execute a number of queries in one block using single command.
3. One can create a PL/SQL unit such as procedures, functions, packages, triggers, and types, which are stored in the database for reuse by applications.
4. PL/SQL provides a feature to handle the exception which occurs in PL/SQL block known as exception handling block.
5. Applications written in PL/SQL are portable to computer hardware or operating system where Oracle is operational.
6. PL/SQL Offers extensive error checking.

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.
Cannot contain PL/SQL code in it.	It is an extension of SQL, so it can contain SQL inside it.

Structure of PL/SQL Block:

PL/SQL extends SQL by adding constructs found in procedural languages, resulting in a structural language that is more powerful than SQL. The basic unit in PL/SQL is a block. All PL/SQL programs are made up of blocks, which can be nested within each other.



Typically, each block performs a logical action in the program. A block has the following structure:

```
DECLARE
    declaration statements;

BEGIN
    executable statements

EXCEPTIONS
    exception handling statements

END;
```

- Declare section starts with **DECLARE** keyword in which variables, constants, records as cursors can be declared which stores data temporarily. It basically consists definition of PL/SQL identifiers. This part of the code is optional.
- Execution section starts with **BEGIN** and ends with **END** keyword.This is a mandatory section and here the program logic is written to perform any task like loops and conditional statements. It supports all DML commands, DDL commands and SQL*PLUS built-in functions as well.
- Exception section starts with **EXCEPTION** keyword.This section is optional which contains statements that are executed when a run-time error occurs. Any exceptions can be handled in this section.

PL/SQL BLOCKS

- In PL/SQL, All statements are classified into units that is called Blocks.
- PL/SQL blocks can include variables, SQL statements, loops, constants, conditional statements and exception handling.
- Blocks can also build a function or a procedure or a package.

PL/SQL blocks are two types:

- Anonymous blocks
- Named blocks

- **1. Anonymous blocks:** In PL/SQL, That's blocks which is not have header are known as anonymous blocks. These blocks do not form the body of a function or triggers or procedure. Example: Here a code example of find greatest number with Anonymous blocks.

```
DECLARE
    -- declare variable a, b and c
    -- and these three variables datatype are integer
   a number;
   b number;
   c number;
BEGIN
   a:= 10;
   b:= 100;
    --find largest number
    --take it in c variable
   IF a > b THEN
      c:= a;
   ELSE
      c:= b;
   END IF;
   dbms_output.put_line(' Maximum number in 10 and 100: ' || c);
END;
/
-- Program End
```

- **2. Named blocks:** That's PL/SQL blocks which having header or labels are known as Named blocks. These blocks can either be subprograms like functions, procedures, packages or Triggers. Example: Here a code example of find greatest number with Named blocks means using function.

```
DECLARE
   a number;
   b number;
   c number;
   --Function return largest number of
   -- two given 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:= 10;
   b:= 100;
   c := findMax(a, b);
   dbms_output.put_line(' Maximum number in 10 and 100 is: ' || c);
END;
/
-- Program End
```

PL/SQL Control Structure

1. IF-THEN-ELSE Statements:

- The IF statement is used to execute a block of code if a condition is true.
- The ELSE statement is used to execute a different block of code if the condition is false.

Syntax:

```
IF condition THEN
  -- Code to execute if the condition is true.
ELSIF another_condition THEN
  -- Code to execute if another_condition is true.
ELSE
  -- Code to execute if neither condition is true.
END IF;
```

Example:

```
DECLARE
  x NUMBER := 10;
BEGIN
  IF x > 5 THEN
    DBMS_OUTPUT.PUT_LINE('x is greater than 5');
  ELSE
    DBMS_OUTPUT.PUT_LINE('x is not greater than
5');
  END IF;
END;
```

2. CASE Statements:

- The CASE statement allows you to perform conditional branching based on the value of an expression.
- You can use the simple CASE or the searched CASE.

Syntax:

```
CASE expression
  WHEN value1 THEN
    -- Code to execute if expression equals value1.
  WHEN value2 THEN
    -- Code to execute if expression equals value2.
  ELSE
    -- Code to execute if none of the values match expression.
END CASE;
```

Example:

```
DECLARE
  day NUMBER := 3;
BEGIN
  CASE day
    WHEN 1 THEN
      DBMS_OUTPUT.PUT_LINE('Monday');
    WHEN 2 THEN
      DBMS_OUTPUT.PUT_LINE('Tuesday');
    WHEN 3 THEN
      DBMS_OUTPUT.PUT_LINE('Wednesday');
    ELSE
      DBMS_OUTPUT.PUT_LINE('Other day');
  END CASE;
END;
```

3. LOOPING Statements:

- PL/SQL provides several types of loops, including simple loops, FOR loops, and WHILE loops.
- Loops are used to repeat a block of code until a specific condition is met.

i.) Simple Loop

LOOP

-- Code to execute repeatedly.

EXIT WHEN condition;

-- Exit the loop when the condition is true.

END LOOP;

Example:

```
DECLARE
counter NUMBER := 1;
BEGIN
    LOOP
        -- Code to execute repeatedly.
    EXIT WHEN condition;
        -- Exit when the condition is true.
    END LOOP;
END;
```

ii.) For Loop:

```
FOR counter IN 1..10 LOOP
    -- Code to execute.
END LOOP;
```

Example:

```
DECLARE
counter NUMBER := 1;
BEGIN
    FOR i IN 1..5 LOOP
DBMS_OUTPUT.PUT_LINE('FOR Loop Iteration: '
|| i);
    END LOOP;
END;
```

iii.) While Loop:

```
WHILE condition LOOP
    -- Code to execute.
END LOOP;
```

Example:

```
DECLARE
counter NUMBER := 1;
BEGIN
    WHILE counter > 0 LOOP
DBMS_OUTPUT.PUT_LINE('WHILE Loop
Iteration: ' || counter);
    counter := counter - 1;
    END LOOP;
END;
```

4. Exit Statement:

EXIT statements are used to exit loops or blocks of code prematurely based on a specified condition.

Syntax:

```
EXIT [WHEN condition];
```

Example:

```
DECLARE
  total NUMBER := 0;
BEGIN
  FOR i IN 1..10 LOOP
    total := total + i;
    EXIT WHEN total > 15;
  END LOOP;
  DBMS_OUTPUT.PUT_LINE('Total: ' || total);
END;
```

PL/SQL Functions

Function

- A Function is a self-contained block of statement that can be used for many times by calling its name to performed well defined task.

PL/SQL Functions

- Functions are named PL/SQL block which means they can be stored into the database as a database object and can be reused.

Syntax:

```
CREATE OR REPLACE FUNCTION function_name
(parmeter1, parameter2….)
RETURN datatype
IS
    Declare variable, constant etc.,
BEGIN
    Executable Statements
    Return (Return Value);
END;
```

```
DECLARE
   a number;
   b number;
   c number;
   --Function return largest number of
   -- two given 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:= 10;
   b:= 100;
   c := findMax(a, b);
   dbms_output.put_line(' Maximum number in 10 and 100 is: ' || c);
END;
/
-- Program End
```

PL/SQL Stored Procedure

- A Stored Procedure is a self-contained subprogram that is meant to do some specific task.
- Procedures are named PL/SQL blocks thus they can be reused, because they are stored into the database as database object.

SYNTAX:

```
CREATE OR REPLACE PROCEDURE
pro_name(para1, para2,…)
AS
    Declaration Statements
BEGIN
    Executable Statements
END Procedure name;
/
```

```
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;
/
```

Stored Procedure

- A stored procedure is a prepared SQL code that you can save, so the code can be reused over and over again.
- So if you have an SQL query that you write over and over again, save it as a stored procedure, and then just call it to execute it.
- You can also pass parameters to a stored procedure, so that the stored procedure can act based on the parameter value(s) that is passed.

- CREATE PROCEDURE *procedure_name*
  AS
  sql_statement
  GO;

Execute a Stored Procedure

- EXEC *procedure_name*;

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico
4	Around the Horn	Thomas Hardy	120 Hanover Sq.	London	WA1 1DP	UK
5	Berglunds snabbköp	Christina Berglund	Berguvsvägen 8	Luleå	S-958 22	Sweden

The following SQL statement creates a stored procedure that selects Customers details from the "Customers" table:

- CREATE PROCEDURE SelectAllCustomers
  AS
  SELECT * FROM Customers
  GO;

- EXEC SelectAllCustomers;

The following SQL statement creates a stored procedure that selects Customers from a particular City from the "Customers" table:

- CREATE PROCEDURE SelectAllCustomers
  @City nvarchar(30)
  AS
  SELECT * FROM Customers WHERE City = @City
  GO;

- EXEC SelectAllCustomers @City = 'London';

Multiple paramaters

- CREATE PROCEDURE SelectAllCustomers
  @City nvarchar(30), @PostalCode
  nvarchar(10)
  AS
  SELECT * FROM Customers WHERE City =
  @City AND PostalCode = @PostalCode
  GO;

- EXEC SelectAllCustomers @City = 'London',
  @PostalCode = 'WA1 1DP';

TRIGGERS

Definition

- In SQL (Structured Query Language), a trigger is a database object that is associated with a specific table and is designed to automatically execute a set of actions or statements in response to certain events or conditions occurring within that table. Triggers are used to enforce data integrity, perform auditing, and automate tasks when certain database events occur.

- There are two main types of triggers in SQL:

BEFORE Trigger: This type of trigger is executed before an event (such as an INSERT, UPDATE, or DELETE operation) takes place. It can be used to validate data or modify it before it is actually written to the table.

AFTER Trigger: An AFTER trigger is executed after an event has occurred. It is often used for tasks like logging changes to the data, sending notifications, or performing calculations based on the updated data.

Events That Trigger Triggers:

Triggers are associated with specific events on a table, such as:

- INSERT: Triggered when a new row is inserted into the table.
- UPDATE: Triggered when an existing row is updated.
- DELETE: Triggered when a row is deleted from the table.

Syntax:

```
CREATE [OR REPLACE] TRIGGER trigger_name
[BEFORE | AFTER] event
ON table_name
FOR EACH ROW
[WHEN (condition)]
BEGIN
   -- Trigger logic here
END;
```

1. **trigger_name:** The name of the trigger.
2. **BEFORE or AFTER:** Specifies whether the trigger should fire before or after the specified event.
3. **event:** The event that activates the trigger (e.g., INSERT, UPDATE, or DELETE).
4. **table_name:** The name of the table associated with the trigger.
5. **FOR EACH ROW:** Indicates that the trigger should execute for each row affected by the event.
6. **WHEN (condition):** An optional condition that can be used to further filter when the trigger should activate.
7. **BEGIN and END:** The block of SQL statements that define the actions to be taken when the trigger fires.

1. Indicate when new record is inserted:

```
CREATE OR REPLACE TRIGGER stu_insert_trigger
AFTER INSERT ON student
FOR EACH ROW
BEGIN
    DBMS_OUTPUT.PUT_LINE('New student record inserted with Student ID: ' || :NEW.student_id);
END;
/
```

2. Indicate before updating the data from the table:

```
CREATE OR REPLACE TRIGGER
before_update_stu_trigger
BEFORE UPDATE ON student
FOR EACH ROW
BEGIN
    DBMS_OUTPUT.PUT_LINE('Updating student
record with Student ID: ' || :OLD.student_id);
END;
```

3. Indicate before deleting the data

```
CREATE OR REPLACE TRIGGER
before_delete_student_trigger
BEFORE DELETE ON student
FOR EACH ROW
BEGIN
    DBMS_OUTPUT.PUT_LINE('Deleting student
record with Student ID: ' || :OLD.student_id);
END;
/
```

4. Indicate after the deletion:

```
CREATE OR REPLACE TRIGGER
after_delete_student_trigger
AFTER DELETE ON student
FOR EACH ROW
BEGIN
    DBMS_OUTPUT.PUT_LINE('Student record
with Student ID: ' || :OLD.student_id || ' has
been deleted.');
END;
```