PLSQL Notes

1. **Welcome**
    - Welcome to the PL/SQL course! This course will guide you through the basics and advanced topics of PL/SQL.
2. **How to make best use of the Program**
    - To make the best use of this program, ensure you have a good understanding of SQL. Practice the examples and exercises provided in each topic.
3. **Installing Oracle**
    - Oracle can be installed from the official Oracle website. Choose the correct version for your operating system. Follow the installation wizard steps.
4. **Installing Java SDK**
    - Java SDK is required for running SQL Developer. It can be downloaded from the official Oracle website. After downloading, run the installer and follow the prompts to install.
5. **Installing SQL Developer**
    - SQL Developer is an IDE for working with Oracle Database. It can be downloaded from the Oracle website. After downloading, unzip the file and run the SQL Developer executable.
6. **Running scripts necessary for the course**
    - Scripts for creating tables, inserting data, etc., will be provided throughout the course. These can be run in SQL Developer.
7. **Scripts for our Lab Exercises**
    - Lab exercises will provide hands-on experience with PL/SQL. Scripts for these exercises will be provided.
8. **What is PL/SQL**
    - PL/SQL is a procedural language designed specifically for the seamless processing of SQL commands. It provides a programming language that allows procedural constructs and exception handling along with SQL's capabilities.
9. **PL/SQL Advantages**
    - PL/SQL allows for better productivity, performance, scalability, security and support for SQL.
10. **PL/SQL Structure**
    - A PL/SQL block consists of three sections: a declarative part, an executable part, and an exception-handling part. Only the executable part is required; the declarative and exception-handling parts are optional.

11. **First Example**

Let's start with a simple PL/SQL block that prints 'Hello, World!'.

```
BEGIN
   DBMS_OUTPUT.PUT_LINE('Hello, World!');
END;
```

12. **Declaring Variables**

Variables in PL/SQL are declared using the `DECLARE` keyword. For example, to declare a variable `x` of type `NUMBER`, you would write:

```
DECLARE
   x NUMBER;
BEGIN
   x := 10;
   DBMS_OUTPUT.PUT_LINE('The value of x is ' || x);
END;
```

13. **Comments in PL/SQL**

Comments can be added using `--` for single line comments and `/* ... */` for multi-line comments.

```
-- This is a single line comment
/*
This is a
multi-line comment
*/
```

14. **Scope of Variables**
    - The scope of a variable is the region of the program where the variable can be referenced. Variables declared in a block are only accessible within that block.

15. **IF then ELSE statement**

The `IF-THEN-ELSE` statement allows for conditional execution of code.

```
DECLARE
   x NUMBER := 10;
BEGIN
   IF x > 20 THEN
      DBMS_OUTPUT.PUT_LINE('x is greater than 20');
   ELSE
      DBMS_OUTPUT.PUT_LINE('x is not greater than 20');
   END IF;
END;
```

16. **CASE Statement**

The `CASE` statement allows for conditional execution of code based on the value of an expression.

```
DECLARE
   x NUMBER := 10;
   result VARCHAR2(20);
BEGIN
   CASE
      WHEN x > 20 THEN result := 'x is greater than 20';
      ELSE result := 'x is not greater than 20';
   END CASE;
   DBMS_OUTPUT.PUT_LINE(result);
END;
```

17. **WHILE Loop**

The `WHILE` loop executes a block of code as long as a condition is true.

```
DECLARE
   x NUMBER := 0;
BEGIN
   WHILE x < 10 LOOP
      DBMS_OUTPUT.PUT_LINE('x is ' || x);
      x := x + 1;
   END LOOP;
END;
```

18. **FOR Loop**

The `FOR` loop executes a block of code a specified number of times.

```
DECLARE
BEGIN
   FOR i IN 1..10 LOOP
      DBMS_OUTPUT.PUT_LINE('i is ' || i);
   END LOOP;
END;
```

19. **Exercise #1**

Try to write a PL/SQL block that declares two variables, `a` and `b`, assigns them values, and then swaps their values without using a third variable.

20. **Reading data from database**

You can read data from a database using a `SELECT` statement inside a PL/SQL block.

```
DECLARE
   emp_name VARCHAR2(100);
BEGIN
   SELECT first_name INTO emp_name FROM employees WHERE
employee_id = 100;
   DBMS_OUTPUT.PUT_LINE('Employee name is ' || emp_name);
END;
```

21. **What is %TYPE**

The `%TYPE` attribute provides the datatype of a variable or a database column. This is particularly useful when declaring variables that will hold database values.

```
DECLARE
   emp_id employees.employee_id%TYPE;
BEGIN
   SELECT employee_id INTO emp_id FROM employees WHERE
first_name = 'John';
   DBMS_OUTPUT.PUT_LINE('Employee ID is ' || emp_id);
END;
```

22. **Inserting data into database**

You can insert data into a database using the `INSERT INTO` statement.

```
DECLARE
   emp_id employees.employee_id%TYPE := 200;
   emp_name employees.first_name%TYPE := 'John';
BEGIN
   INSERT INTO employees (employee_id, first_name) VALUES
(emp_id, emp_name);
   COMMIT;
END;
```

23. **Exercise #2**

Try to write a PL/SQL block that inserts a new row into the `employees` table with an `employee_id` of 200 and a `first_name` of 'John'.

24. **Anonymous Blocks**

An anonymous block is a PL/SQL block that appears within your application and it is not named or stored in the database. The previous examples are all anonymous blocks.

25. **What are Procedures?**

A procedure is a named PL/SQL block that performs one or more actions. Procedures are created with the `CREATE PROCEDURE` statement.

```
CREATE PROCEDURE print_emp_name (emp_id IN NUMBER) AS
   emp_name employees.first_name%TYPE;
BEGIN
   SELECT first_name INTO emp_name FROM employees WHERE
employee_id = emp_id;
   DBMS_OUTPUT.PUT_LINE('Employee name is ' || emp_name);
END;
```

26. **Creating a Procedure**

Procedures are created using the `CREATE PROCEDURE` statement. Here's an example of creating a procedure that prints the name of an employee given their ID.

```
CREATE PROCEDURE print_emp_name (emp_id IN NUMBER) AS
   emp_name employees.first_name%TYPE;
BEGIN
   SELECT first_name INTO emp_name FROM employees WHERE
employee_id = emp_id;
   DBMS_OUTPUT.PUT_LINE('Employee name is ' || emp_name);
END;
```

27. **Calling a Procedure**

Once a procedure is created, it can be called using the `EXECUTE` statement or simply the procedure name.

```
BEGIN
   print_emp_name(100);
END;
```

28. **Procedure with OUT Mode**

Procedures can have OUT parameters, which are used to return values. Here's an example of a procedure with an OUT parameter.

```
CREATE PROCEDURE get_emp_name (emp_id IN NUMBER, emp_name
OUT VARCHAR2) AS
BEGIN
   SELECT first_name INTO emp_name FROM employees WHERE
employee_id = emp_id;
END;
```

29. **Procedure with IN OUT Mode**

Procedures can also have IN OUT parameters, which can be used to both pass values into a procedure and return values. Here's an example.

```
CREATE PROCEDURE update_emp_name (emp_id IN OUT NUMBER,
emp_name IN OUT VARCHAR2) AS
BEGIN
   SELECT first_name INTO emp_name FROM employees WHERE
employee_id = emp_id;
   emp_id := emp_id + 1;
END;
```

30. **What are Functions?**

Functions are similar to procedures but are designed to return a single value. Functions are created using the `CREATE FUNCTION` statement.

```
CREATE FUNCTION get_emp_name (emp_id IN NUMBER) RETURN
VARCHAR2 AS
   emp_name employees.first_name%TYPE;
BEGIN
   SELECT first_name INTO emp_name FROM employees WHERE
employee_id = emp_id;
   RETURN emp_name;
END;
```

31. **Calling a Function**

Once a function is created, it can be called in a similar way to a procedure. However, since a function returns a value, it can be used in expressions.

```
DECLARE
   emp_name VARCHAR2(100);
BEGIN
   emp_name := get_emp_name(100);
   DBMS_OUTPUT.PUT_LINE('Employee name is ' || emp_name);
END;
```

32. **Exercise #3**

Try to write a function that takes an `employee_id` as input and returns the `last_name` of the employee.

33. **What are Exceptions**

Exceptions are events that alter the normal flow of execution. PL/SQL provides a feature to handle such exceptions so that normal execution can be continued or meaningful information can be reported back to the user.

```
DECLARE
   emp_id employees.employee_id%TYPE := 9999;
   emp_name employees.first_name%TYPE;
BEGIN
   SELECT first_name INTO emp_name FROM employees WHERE
employee_id = emp_id;
   DBMS_OUTPUT.PUT_LINE('Employee name is ' || emp_name);
EXCEPTION
   WHEN NO_DATA_FOUND THEN
      DBMS_OUTPUT.PUT_LINE('No employee found with ID '
|| emp_id);
END;
```

34. **Exceptions Example**

In the above example, if there is no employee with an ID of 9999, a `NO_DATA_FOUND` exception will be raised. The code in the `EXCEPTION` block will then be executed.

35. **User Defined Exceptions**

Apart from system-defined exceptions, users can define their own exceptions in PL/SQL. User-defined exceptions must be declared and raised explicitly by the programmer.

```
DECLARE
   emp_id employees.employee_id%TYPE := 9999;
   emp_name employees.first_name%TYPE;
   ex_custom EXCEPTION;
BEGIN
   SELECT first_name INTO emp_name FROM employees WHERE
employee_id = emp_id;
   IF emp_name IS NULL THEN
      RAISE ex_custom;
   END IF;
   DBMS_OUTPUT.PUT_LINE('Employee name is ' || emp_name);
```

```
EXCEPTION
   WHEN ex_custom THEN
      DBMS_OUTPUT.PUT_LINE('No employee found with ID '
|| emp_id);
END;
```

36. **System Defined Exceptions list**

PL/SQL provides many pre-defined exceptions, some of which are:

`NO_DATA_FOUND`: This exception is raised when a `SELECT INTO` statement returns no rows.

`TOO_MANY_ROWS`: This exception is raised when a `SELECT INTO` statement returns more than one row.

`ZERO_DIVIDE`: This exception is raised when an attempt is made to divide a number by zero.

`INVALID_NUMBER`: This exception is raised when an attempt is made to convert a string to a number, and the string does not represent a valid number.

37. **Exercise #4**

Try to write a PL/SQL block that handles the `ZERO_DIVIDE` exception.

38. **What are Packages**

A package is a schema object that groups logically related PL/SQL types, variables, constants, subprograms, cursors, and exceptions. A package is compiled and stored in the database, where many applications can share its contents.

39. **Package Specification**

The package specification contains the public declarations. The scope of these declarations is public to the package and any subprograms that invoke the package.

40. **Package Body**

The package body contains the implementation of the methods declared in the package specification, as well as the definitions of any private data types and private subprograms.

41. **Executing sub programs present in Packages**

Once a package is created, its subprograms can be called using the package name and the subprogram name.

```
BEGIN
   package_name.subprogram_name(parameters);
END;
```

42. **Exercise #5**

Try to write a package that contains a procedure to insert a new employee into the `employees` table and a function to retrieve an employee's name given their ID.

43. **What are Records?**

A record is a composite data structure that allows you to store different types of information in the same variable. In PL/SQL, you can define a record type using the `TYPE` keyword.

```
DECLARE
   TYPE employee_record IS RECORD (
      id employees.employee_id%TYPE,
      name employees.first_name%TYPE
   );
   emp employee_record;
BEGIN
   SELECT employee_id, first_name INTO emp FROM employees
WHERE employee_id = 100;
   DBMS_OUTPUT.PUT_LINE('Employee name is ' || emp.name);
END;
```

44. **Records Example**

In the above example, a record type `employee_record` is declared, which can hold an employee ID and name. A variable `emp` of type `employee_record` is then declared, and a row from the `employees` table is selected into this record.

45. **Working with Record data**

Once you have a record, you can access its fields using the dot notation.

```
DECLARE
   TYPE employee_record IS RECORD (
      id employees.employee_id%TYPE,
      name employees.first_name%TYPE
   );
   emp employee_record;
BEGIN
   SELECT employee_id, first_name INTO emp FROM employees
WHERE employee_id = 100;
   DBMS_OUTPUT.PUT_LINE('Employee ID is ' || emp.id);
   DBMS_OUTPUT.PUT_LINE('Employee name is ' || emp.name);
END;
```

46. **Passing Records as parameters**

Records can be passed as parameters to procedures and functions.

```
CREATE PROCEDURE print_emp(emp IN employee_record) AS
BEGIN
   DBMS_OUTPUT.PUT_LINE('Employee ID is ' || emp.id);
   DBMS_OUTPUT.PUT_LINE('Employee name is ' || emp.name);
END;
```

47. **Inserting data using Records**

You can insert data into a table using a record. The fields in the record should match the columns in the table.

```
DECLARE
   TYPE employee_record IS RECORD (
      id employees.employee_id%TYPE,
      name employees.first_name%TYPE
   );
```

```
   emp employee_record;
BEGIN
   emp.id := 200;
   emp.name := 'John';
   INSERT INTO employees (employee_id, first_name) VALUES
emp.id, emp.name;
   COMMIT;
END;
```

48. **Updating data using Records**

Similarly, you can update data in a table using a record.

```
DECLARE
   TYPE employee_record IS RECORD (
      id employees.employee_id%TYPE,
      name employees.first_name%TYPE
   );
   emp employee_record;
BEGIN
   emp.id := 200;
   emp.name := 'John';
   UPDATE employees SET first_name = emp.name WHERE
employee_id = emp.id;
   COMMIT;
END;
```

49. **User defined Record Types**

You can define your own record types in PL/SQL. This is done using the `TYPE` keyword.

```
DECLARE
   TYPE employee_record IS RECORD (
      id NUMBER,
      name VARCHAR2(100)
   );
   emp employee_record;
BEGIN
   emp.id := 200;
   emp.name := 'John';
   DBMS_OUTPUT.PUT_LINE('Employee ID is ' || emp.id);
   DBMS_OUTPUT.PUT_LINE('Employee name is ' || emp.name);
END;
```

50. **User defined Record Example**

In the above example, a user-defined record type `employee_record` is declared, which can hold an employee ID and name. A variable `emp` of type `employee_record` is then declared, and values are assigned to its fields.

51. **Exercise #6**

Try to write a PL/SQL block that declares a user-defined record type for an employee, assigns values to its fields, and then prints these values.

52. **What are Cursors?**

A cursor is a database object used to retrieve multiple rows from a database. Cursors are declared using the `CURSOR` keyword.

```
DECLARE
   CURSOR emp_cursor IS SELECT * FROM employees;
   emp emp_cursor%ROWTYPE;
BEGIN
   OPEN emp_cursor;
   LOOP
      FETCH emp_cursor INTO emp;
      EXIT WHEN emp_cursor%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE('Employee name is ' ||
emp.first_name);
   END LOOP;
   CLOSE emp_cursor;
END;
```

53. **Implicit Cursor**

Implicit cursors are automatically created by Oracle when an SQL statement is executed, when there is a need to compute a set of records on more than one row. However, developers have no control over an implicit cursor and the information it holds.

```
DECLARE
   emp_name employees.first_name%TYPE;
BEGIN
   SELECT first_name INTO emp_name FROM employees WHERE
employee_id = 100;
   DBMS_OUTPUT.PUT_LINE('Employee name is ' || emp_name);
```

```
END;
```

54. **Explicit Cursor**

Explicit cursors are programmer-defined cursors for gaining more control over the context area. An explicit cursor should be defined in the declaration section of the PL/SQL Block. It is created on a SELECT Statement which returns more than one row.

```
DECLARE
   CURSOR emp_cursor IS SELECT * FROM employees;
   emp emp_cursor%ROWTYPE;
BEGIN
   OPEN emp_cursor;
   LOOP
      FETCH emp_cursor INTO emp;
      EXIT WHEN emp_cursor%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE('Employee name is ' ||
emp.first_name);
   END LOOP;
   CLOSE emp_cursor;
END;
```

55. **Retrieving 1 row using Explicit Cursor**

You can retrieve a single row from a cursor by using the `FETCH` statement.

```
DECLARE
   CURSOR emp_cursor IS SELECT * FROM employees WHERE
employee_id = 100;
   emp emp_cursor%ROWTYPE;
BEGIN
   OPEN emp_cursor;
   FETCH emp_cursor INTO emp;
   DBMS_OUTPUT.PUT_LINE('Employee name is ' ||
emp.first_name);
   CLOSE emp_cursor;
END;
```

56. **Retrieving more than 1 row using Explicit Cursor**

You can retrieve multiple rows from a cursor by using a loop with the `FETCH` statement.

```
DECLARE
   CURSOR emp_cursor IS SELECT * FROM employees;
   emp emp_cursor%ROWTYPE;
BEGIN
   OPEN emp_cursor;
   LOOP
      FETCH emp_cursor INTO emp;
      EXIT WHEN emp_cursor%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE('Employee name is ' ||
emp.first_name);
   END LOOP;
   CLOSE emp_cursor;
END;
```

57. **Using Records in Cursors**

You can fetch rows into a record from a cursor. The record must be of the same type as the cursor.

```
DECLARE
   CURSOR emp_cursor IS SELECT * FROM employees;
   emp emp_cursor%ROWTYPE;
BEGIN
   OPEN emp_cursor;
   FETCH emp_cursor INTO emp;
   DBMS_OUTPUT.PUT_LINE('Employee name is ' ||
emp.first_name);
   CLOSE emp_cursor;
END;
```

58. **Cursor FOR Loop**

A cursor FOR loop implicitly declares %ROWTYPE as loop index, opens a cursor, fetches rows of values from the cursor into fields in the record, and closes the cursor when all rows have been processed.

```
DECLARE
   CURSOR emp_cursor IS SELECT * FROM employees;
BEGIN
   FOR emp IN emp_cursor LOOP
      DBMS_OUTPUT.PUT_LINE('Employee name is ' ||
emp.first_name);
   END LOOP;
END;
```

59. **Cursor Variable and Reference Cursor**

A cursor variable is a type of data structure that can hold a cursor. A cursor variable is more flexible because it is not tied to a specific query. You can open a cursor variable for any query that returns the right set of columns.

```
DECLARE
   TYPE ref_cursor IS REF CURSOR;
   emp_cursor ref_cursor;
   emp_rec employees%ROWTYPE;
BEGIN
   OPEN emp_cursor FOR SELECT * FROM employees;
   LOOP
      FETCH emp_cursor INTO emp_rec;
      EXIT WHEN emp_cursor%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE('Employee name is ' ||
emp_rec.first_name);
   END LOOP;
   CLOSE emp_cursor;
END;
```

60. **Exceptions for Cursors**

Exceptions can be raised during cursor processing. For example, if you fetch from a cursor after the last row has been retrieved, you can catch the `NO_DATA_FOUND` exception.

```
DECLARE
   CURSOR emp_cursor IS SELECT * FROM employees;
   emp_rec employees%ROWTYPE;
BEGIN
   OPEN emp_cursor;
   LOOP
      FETCH emp_cursor INTO emp_rec;
      EXIT WHEN emp_cursor%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE('Employee name is ' ||
emp_rec.first_name);
   END LOOP;
   FETCH emp_cursor INTO emp_rec;  -- This will raise an
exception
   CLOSE emp_cursor;
EXCEPTION
   WHEN NO_DATA_FOUND THEN
      DBMS_OUTPUT.PUT_LINE('No more data to fetch');
END;
```

61. **Exercise #7**

Try to write a PL/SQL block that declares a cursor for the `employees` table, fetches rows from this cursor into a record, and prints the employee names. Handle the `NO_DATA_FOUND` exception to gracefully exit the loop when all rows have been fetched.

62. **What are Collections**

Collections are single-dimensional arrays that can be accessed by an index. They can be used to store multiple values of the same type. PL/SQL provides three types of collections: Associative arrays, Nested tables, and VARRAYs.

63. **Collection Terminology**

Some terms related to collections are:

Element: Each item in a collection.

Index: The position of an element in a collection.

Collection Methods: Built-in functions and procedures that operate on collections.

64. **Associative Arrays**

Associative arrays (also known as index-by tables) are sets of key-value pairs. Each key is unique and is used to locate the corresponding value.

```
DECLARE
   TYPE name_array IS TABLE OF VARCHAR2(100) INDEX BY
PLS_INTEGER;
   names name_array;
BEGIN
   names(1) := 'John';
   names(2) := 'Jane';
   DBMS_OUTPUT.PUT_LINE('Name 1 is ' || names(1));
   DBMS_OUTPUT.PUT_LINE('Name 2 is ' || names(2));
END;
```

65. **Nested Tables**

Nested tables are similar to index-by tables, but they do not have a primary key. Elements are stored in no particular order.

```
DECLARE
   TYPE name_table IS TABLE OF VARCHAR2(100);
   names name_table := name_table('John', 'Jane');
BEGIN
   FOR i IN names.FIRST..names.LAST LOOP
      DBMS_OUTPUT.PUT_LINE('Name ' || i || ' is ' ||
names(i));
   END LOOP;
END;
```

66. **What are VARRAYS**

VARRAYs (variable-size arrays) are arrays that have a maximum size. You must specify the maximum size when you declare the VARRAY. The size cannot be changed after the VARRAY is declared.

```
DECLARE
   TYPE name_varray IS VARRAY(2) OF VARCHAR2(100);
   names name_varray := name_varray('John', 'Jane');
BEGIN
   FOR i IN 1..names.COUNT LOOP
      DBMS_OUTPUT.PUT_LINE('Name ' || i || ' is ' ||
names(i));
   END LOOP;
END;
```

67. **Collection Methods**

Some methods provided by collections are:

COUNT: Returns the number of elements in the collection.

EXISTS: Returns `TRUE` if the specified element exists in the collection.

FIRST and LAST: Return the first and last (smallest and largest) index numbers in the collection.

68. **MULTISET Operators**

MULTISET operators allow you to combine nested tables into a single nested table. They include `MULTISET UNION`, `MULTISET INTERSECT`, and `MULTISET EXCEPT`.

```
DECLARE
   TYPE name_table IS TABLE OF VARCHAR2(100);
   names1 name_table := name_table('John', 'Jane');
   names2 name_table := name_table('Jane', 'Joe');
   all_names name_table;
BEGIN
   all_names := names1 MULTISET UNION names2;
   FOR i IN all_names.FIRST..all_names.LAST LOOP
      DBMS_OUTPUT.PUT_LINE('Name ' || i || ' is ' ||
all_names(i));
   END LOOP;
```

```
END;
```

69. **Collections Summary**

Collections are a powerful feature in PL/SQL. They allow you to handle sets of data more efficiently. You can choose from associative arrays, nested tables, and VARRAYs depending on your needs.

70. **Exercise #8**

Try to write a PL/SQL block that declares a nested table of numbers, inserts some numbers into it, and then prints these numbers.

71. **Scripts necessary to practise Triggers**

Triggers are stored programs, which are automatically executed or fired when some events occur. Triggers are, in fact, written to be executed in response to any of the following events −

A database manipulation (DML) statement (DELETE, INSERT, or UPDATE).

A database definition (DDL) statement (CREATE, ALTER, or DROP).

A database operation (SERVERERROR, LOGON, LOGOFF, STARTUP, or SHUTDOWN).

72. **Trigger's Introduction**

A trigger is a special kind of stored procedure that automatically executes when an event occurs in the database server. DML triggers execute when a user tries to modify data through a data manipulation language (DML) event. DML events are INSERT, UPDATE, or DELETE statements on a table or view.

73. **Trigger Types**

There are two types of triggers based on the level of trigger action: row level trigger and statement level trigger.

Row Level Trigger: An event is triggered for each row updated, inserted or deleted.

Statement Level Trigger: An event is triggered for each SQL statement executed.

74. **Statement Level Trigger**

Statement-level triggers are fired once for each transaction. For example, if an UPDATE statement updates multiple rows of a table, a statement-level trigger on that table is fired only once.

```
CREATE OR REPLACE TRIGGER before_update_salary
BEFORE UPDATE OF salary ON employees
BEGIN
   DBMS_OUTPUT.PUT_LINE('Updating salary...');
END;
```

75. **Statement Level Trigger with multiple Actions**

A statement level trigger can perform multiple actions. For example, it can update multiple columns or tables.

```
CREATE OR REPLACE TRIGGER before_update_salary
BEFORE UPDATE OF salary ON employees
FOR EACH ROW
BEGIN
   :NEW.salary := :NEW.salary * 1.1;
   :NEW.last_update := SYSDATE;
END;
```

76. **Row Level Trigger**

Row-level triggers execute once for each row in a transaction. For example, if an UPDATE statement updates multiple rows of a table, a row-level trigger on that table is fired once for each row affected by the UPDATE statement.

```
CREATE OR REPLACE TRIGGER before_update_salary
BEFORE UPDATE OF salary ON employees
FOR EACH ROW
BEGIN
   :NEW.salary := :NEW.salary * 1.1;
END;
```

77. **OLD and NEW Pseudo Records with an Example**

In a row-level trigger, the `:OLD` and `:NEW` pseudo records are used to reference the old and new values of the row being processed.

```
CREATE OR REPLACE TRIGGER before_update_salary
BEFORE UPDATE OF salary ON employees
FOR EACH ROW
BEGIN
   DBMS_OUTPUT.PUT_LINE('Old salary was ' ||
:OLD.salary);
   :NEW.salary := :NEW.salary * 1.1;
   DBMS_OUTPUT.PUT_LINE('New salary is ' || :NEW.salary);
END;
```

78. **Restricting the Trigger based on a Condition using the WHEN clause**

You can restrict the firing of a trigger based on a condition using the `WHEN` clause.

```
CREATE OR REPLACE TRIGGER before_update_salary
BEFORE UPDATE OF salary ON employees
FOR EACH ROW
WHEN (NEW.salary > 10000)
BEGIN
   :NEW.salary := :NEW.salary * 1.1;
END;
```

79. **Restricting the Trigger at a column level using the OF clause**

You can restrict the firing of a trigger to specific columns using the `OF` clause.

```
CREATE OR REPLACE TRIGGER before_update_salary
BEFORE UPDATE OF salary ON employees
FOR EACH ROW
BEGIN
   :NEW.salary := :NEW.salary * 1.1;
END;
```

In this example, the trigger will only fire when the `salary` column is updated.

80. **Enable/Disable/Drop a Trigger**

You can enable, disable, or drop a trigger using the `ALTER TRIGGER` and `DROP TRIGGER` statements.

```
ALTER TRIGGER before_update_salary DISABLE;
ALTER TRIGGER before_update_salary ENABLE;
DROP TRIGGER before_update_salary;
```

81. **Context Switch**

A context switch occurs when the system changes from running one PL/SQL program unit to running another. This can occur when a program calls a subprogram, or when a trigger fires in response to a triggering event.

82. **Bulk Processing Introduction**

Bulk processing is a method of processing high volumes of data where a group of SQL statements are collected, and then it is sent to database for the further processing. It increases the performance and reduces the context switch between the SQL and PL/SQL engine.

83. **Bulk Processing Example**

Here's an example of bulk processing using the `FORALL` statement.

```
DECLARE
   TYPE emp_tab IS TABLE OF employees%ROWTYPE INDEX BY
PLS_INTEGER;
   emps emp_tab;
```

```
BEGIN
   SELECT * BULK COLLECT INTO emps FROM employees;
   FORALL i IN 1..emps.COUNT
      UPDATE employees SET salary = salary * 1.1 WHERE
employee_id = emps(i).employee_id;
   COMMIT;
END;
```

84. **Bulk Processing with LIMIT option**

The `LIMIT` option can be used with `BULK COLLECT` to limit the number of rows fetched.

```
DECLARE
   CURSOR emp_cursor IS SELECT * FROM employees;
   TYPE emp_tab IS TABLE OF emp_cursor%ROWTYPE;
   emps emp_tab;
BEGIN
   OPEN emp_cursor;
   LOOP
      FETCH emp_cursor BULK COLLECT INTO emps LIMIT 100;
      EXIT WHEN emps.COUNT = 0;
      -- process batch here
   END LOOP;
   CLOSE emp_cursor;
END;
```

85. **Bulk Processing with ROWTYPE**

You can use `%ROWTYPE` with `BULK COLLECT` to fetch rows into a collection of records.

```
DECLARE
   TYPE emp_tab IS TABLE OF employees%ROWTYPE INDEX BY
PLS_INTEGER;
   emps emp_tab;
BEGIN
   SELECT * BULK COLLECT INTO emps FROM employees;
   FOR i IN 1..emps.COUNT LOOP
```

```
      DBMS_OUTPUT.PUT_LINE('Employee name is ' ||
emps(i).first_name);
   END LOOP;
END;
```

86. **Handling Bulk Processing Exceptions**

Exceptions can be handled during bulk processing using the `SAVE EXCEPTIONS` clause with `FORALL` and the `SQL%BULK_EXCEPTIONS` attribute.

```
DECLARE
   TYPE emp_tab IS TABLE OF NUMBER INDEX BY PLS_INTEGER;
   emps emp_tab;
   ex_dml_errors EXCEPTION;
   PRAGMA EXCEPTION_INIT(ex_dml_errors, -24381);
BEGIN
   SELECT employee_id BULK COLLECT INTO emps FROM
employees;
   FORALL i IN 1..emps.COUNT SAVE EXCEPTIONS
      UPDATE employees SET salary = salary * 1.1 WHERE
employee_id = emps(i);
EXCEPTION
   WHEN ex_dml_errors THEN
      FOR i IN 1..SQL%BULK_EXCEPTIONS.COUNT LOOP
         DBMS_OUTPUT.PUT_LINE('Error ' ||
SQL%BULK_EXCEPTIONS(i).ERROR_INDEX || ' occurred during
bulk operation');
      END LOOP;
END;
```

87. **Dynamic SQL Introduction**

Dynamic SQL is a programming technique that enables you to build SQL statements dynamically at runtime. It allows you to create more general purpose and flexible SQL statement because you can use variables to manipulate the SQL.

88. **Dynamic SQL Example**

Here's an example of dynamic SQL using the `EXECUTE IMMEDIATE` statement.

```
DECLARE
   sql_stmt VARCHAR2(200);
   emp_id NUMBER := 100;
   emp_name VARCHAR2(100);
BEGIN
   sql_stmt := 'SELECT first_name FROM employees WHERE
employee_id = :id';
   EXECUTE IMMEDIATE sql_stmt INTO emp_name USING emp_id;
   DBMS_OUTPUT.PUT_LINE('Employee name is ' || emp_name);
END;
```

89. **Bind variables and Parsing**

Bind variables are used to replace placeholders in a SQL statement. Bind variables can improve performance by allowing the database to reuse the parsed form of the SQL statement.

90. **Dynamic SQL with Bind variables**

Here's an example of dynamic SQL with bind variables.

```
DECLARE
   sql_stmt VARCHAR2(200);
   emp_id NUMBER := 100;
   emp_name VARCHAR2(100);
BEGIN
     sql_stmt := 'SELECT
```

91. **Dynamic SQL with Cursors**

Cursors can be used in dynamic SQL to fetch multiple rows. Here's an example:

```
CREATE OR REPLACE PROCEDURE dynamic_sql (table_name IN
VARCHAR2) AS
   sql_stmt VARCHAR2(200);
   cur SYS_REFCURSOR;
   type_name VARCHAR2(100);
BEGIN
   sql_stmt := 'SELECT typename FROM ' || table_name;
   OPEN cur FOR sql_stmt;
   LOOP
      FETCH cur INTO type_name;
      EXIT WHEN cur%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE('Type name is ' || type_name);
   END LOOP;
   CLOSE cur;
END;
```

92. **Dynamic SQL with Bulk Collect**

`BULK COLLECT` can be used in dynamic SQL to fetch multiple rows into a collection.

```
DECLARE
   TYPE name_tab IS TABLE OF VARCHAR2(100);
   names name_tab;
   sql_stmt VARCHAR2(200);
   cur SYS_REFCURSOR;
BEGIN
   sql_stmt := 'SELECT first_name FROM employees';
   OPEN cur FOR sql_stmt;
   FETCH cur BULK COLLECT INTO names;
   CLOSE cur;
   -- Process the data here
END;
```

93. **DBMS_SQL package**

The `DBMS_SQL` package provides an interface to use dynamic SQL to parse any data manipulation language (DML) or data definition language (DDL) statement using PL/SQL.

```
DECLARE
   cur INTEGER := DBMS_SQL.OPEN_CURSOR;
   rows_processed INTEGER;
BEGIN
   DBMS_SQL.PARSE(cur, 'UPDATE employees SET salary =
salary * 1.1', DBMS_SQL.NATIVE);
   rows_processed := DBMS_SQL.EXECUTE(cur);
   DBMS_SQL.CLOSE_CURSOR(cur);
   DBMS_OUTPUT.PUT_LINE('Rows processed: ' ||
rows_processed);
END;
```

94. **Object Creation**

In Oracle PL/SQL, an object can be a column, a constant, a procedure, a space, a table, or a view that has attributes and behaviors. For example, you can create an object type `Person` with attributes `first_name` and `last_name` and a method `get_full_name`.

```
CREATE TYPE Person AS OBJECT (
   first_name VARCHAR2(100),
   last_name VARCHAR2(100),
   MEMBER FUNCTION get_full_name RETURN VARCHAR2
);
CREATE TYPE BODY Person AS
   MEMBER FUNCTION get_full_name RETURN VARCHAR2 IS
   BEGIN
      RETURN first_name || ' ' || last_name;
   END;
END;
```

95. **Table Functions**

Table functions are functions that produce a collection or rows (either a nested table or a varray) that can be queried like a physical database table. You define a table function to return a nested table datatype, then you can use the TABLE keyword in the FROM clause of a query to operate on the output of the function as if it were a database table.

```
CREATE TYPE numbers_tab IS TABLE OF NUMBER;
CREATE FUNCTION get_numbers RETURN numbers_tab PIPELINED
IS
BEGIN
```

```
   FOR i IN 1..10 LOOP
      PIPE ROW(i);
   END LOOP;
   RETURN;
END;
SELECT * FROM TABLE(get_numbers);
```

96. **Pipelined Table Functions**

Pipelined table functions allow for rows to be returned iteratively, as they are produced. This can provide performance benefits by allowing calling functions to start processing before all the rows are generated.

```
CREATE TYPE numbers_tab IS TABLE OF NUMBER;
CREATE FUNCTION get_numbers RETURN numbers_tab PIPELINED
IS
BEGIN
   FOR i IN 1..10 LOOP
      PIPE ROW(i);
   END LOOP;
   RETURN;
END;
SELECT * FROM TABLE(get_numbers);
```

97. **Large Objects (LOB's) Introduction**

Large Objects (LOBs) are a set of datatypes that are designed to hold large blocks of data. LOBs are used to store semi-structured data (such as text, images, video and audio files) directly in the database.

98. **Character Large Object (CLOB)**

A CLOB (Character Large Object) is used to store unicode character-based data, such as large documents in any character set. The length is specified in number characters for CLOB.

```
DECLARE
   clob_data CLOB;
BEGIN
   clob_data := 'This is some large text data that might
continue for a long length...';
   DBMS_OUTPUT.PUT_LINE('CLOB data: ' || clob_data);
END;
```

99. **Binary Large Object (BLOB)**

A BLOB (Binary Large Object) is used to store binary data such as image files, video files, etc.

```
DECLARE
   blob_data BLOB;
   raw_data RAW(32767);
BEGIN
   raw_data := UTL_RAW.CAST_TO_RAW('This is some raw
data');
   blob_data := TO_BLOB(raw_data);
   DBMS_OUTPUT.PUT_LINE('BLOB data length: ' ||
DBMS_LOB.GETLENGTH(blob_data));
END;
```

100. **Storing Images/Videos in BLOB's** - You can store images or videos in a BLOB column in the database. However, you would typically not do this in a PL/SQL block in your application code. Instead, you would use some form of file upload in your application to accept the file from the user, and then store it in the database.

101. **Binary Files (BFILE)** - A BFILE is a datatype in Oracle PL/SQL that provides read-only access to data located outside the database tablespaces on tertiary storage devices, such as hard disks, network mounted files systems, CD-ROMs, PhotoCDs, and DVDs. BFILE data is not included in database backups.

```plsql
DECLARE
   bfile_loc BFILE;
BEGIN
   bfile_loc := BFILENAME('MY_DIR', 'my_file.txt');
   DBMS_OUTPUT.PUT_LINE('BFILE location: ' ||
GET_LENGTH(bfile_loc));
END;
```

102. **Temporary LOB's** - Temporary LOBs, which are instances of LOB datatypes with a duration of either a database session or a transaction, can be used to manipulate LOBs.

```plsql
DECLARE
   temp_clob CLOB;
BEGIN
   DBMS_LOB.CREATETEMPORARY(temp_clob, TRUE);
   DBMS_LOB.WRITE(temp_clob, 11, 1, 'Hello World');
   DBMS_OUTPUT.PUT_LINE('Temporary CLOB: ' || temp_clob);
   DBMS_LOB.FREETEMPORARY(temp_clob);
END;
```