ORACLE 9i PLSQL

1. Declaring Variables

PL/SQL Block Structure
DECLARE –Optional
Variables, cursors, user-defined exceptions
BEGIN –Mandatory
– SQL statements
– PL/SQL statements
EXCEPTION –Optional
Actions to perform when errors occur
END; –Mandatory

•Section keywords DECLARE, BEGIN, and EXCEPTION are not followed by semicolons.
•END and all other PL/SQL statements require a semicolon to terminate the statement.

Block Types-
Anonymous, Procedure, Function

Types of Variables
• PL/SQL variables:
– Scalar
– Composite
– Reference
– LOB (large objects)
• Non-PL/SQL variables: Bind and host variables

•Scalar data types hold a single value. The main data types are those that correspond to column types in Oracle server tables; PL/SQL also supports Boolean variables.
•Composite data types, such as records, allow groups of fields to be defined and manipulated in PL/SQL blocks.
•Reference data types hold values, called pointers, that designate other program items. Reference data types are not covered in this course.
• LOB data types hold values, called locators, that specify the location of large objects (for example graphic images) that are stored out of line. LOB data types are discussed in detail later in this course.

Using iSQL*Plus Variables Within PL/SQL Blocks
iSQL*Plus host variables can be used to pass run time values out of the PL/SQL block back to the iSQL*Plus environment. You can reference host variables in a PL/SQL block with a preceding colon.

Types of Variables
• TRUE represents a Boolean value.
•25-JAN-01 represents a DATE.
•The photograph represents a BLOB.
•The text of a speech represents a LONG.
•256120.08 represents a NUMBER data type with precision and scale.
•The movie represents a BFILE.

•The city name, Atlanta, represents a VARCHAR2.

Declaring PL/SQL Variables
Syntax:
identifier [CONSTANT] datatype [NOT NULL] [:= | DEFAULT expr];

Guidelines for Declaring PL/SQL Variables
• Follow naming conventions.
• Initialize variables designated as NOT NULL and CONSTANT.
• Declare one identifier per line.
• Initialize identifiers by using the assignment
operator (:=) or the DEFAULT reserved word.
identifier := expr;

Naming Rules
• Two variables can have the same name, provided they are in different blocks.
• The variable name (identifier) should not be the same as the name of table columns used in the block.

Note: The names of the variables must not be longer than 30 characters. The first character must be a letter; the remaining characters can be letters, numbers, or special symbols.

Base Scalar Data Types
• CHAR [(maximum_length)]
• VARCHAR2(maximum_length)
• LONG
• LONG RAW
• NUMBER [(precision, scale)]
• BINARY_INTEGER
• PLS_INTEGER
• BOOLEAN

• DATE
• TIMESTAMP
• TIMESTAMP WITH TIME ZONE
• TIMESTAMP WITH LOCAL TIME ZONE
• INTERVAL YEAR TO MONTH
• INTERVAL DAY TO SECOND

The %TYPE Attribute
• Declare a variable according to:
– A database column definition
– Another previously declared variable
• Prefix %TYPE with:
– The database table and column
– The previously declared variable name

Declaring Boolean Variables
• Only the values TRUE, FALSE, and NULL can be assigned to a Boolean variable.

Composite Data Types
TABLE, RECORD, NESTED TABLE and VARRAY

LOB Data Type Variables
CLOB (character large object) BOOK
BLOB (binary large object) PHOTO
BFILE (binary file) MOVIE
NCLOB (national language character large object)

Using Bind Variables
To reference a bind variable in PL/SQL, you must prefix its name with a colon (:).
Example:

```
VARIABLE g_salary NUMBER
BEGIN
SELECTsalary
INTO:g_salary
FROMemployees
WHEREemployee_id = 178;
END;
/
PRINT g_salary
```

• Variables declared in an external environment such as iSQL*Plus are called host variables.
• Use DBMS_OUTPUT.PUT_LINE to display data from a PL/SQL block.

2. Writing Executable Statements

PL/SQL Block Syntax and Guidelines
• Statements can continue over several lines.
• Lexical units can be classified as:
– Delimiters
– Identifiers
– Literals
– Comments

Identifiers
• Can contain up to 30 characters
• Must begin with an alphabetic character
• Can contain numerals, dollar signs, underscores, and number signs
• Can not contain characters such as hyphens, slashes, and spaces
• Should not have the same name as a database table column name
• Should not be reserved words

Note: Reserved words cannot be used as identifiers unless they are enclosed in double quotation marks (for example, "SELECT").

Commenting Code

• Prefix single-line comments with two dashes (--).
• Place multiple-line comments between the symbols /* and */.

SQL Functions in PL/SQL
• Available in procedural statements:
– Single-row number
– Single-row character
– Data type conversion Same as in SQL
– Date
– Timestamp

– GREATEST and LEAST
– Miscellaneous functions

• Not available in procedural statements:
– DECODE
– Group functions

Identifier Scope
An identifier is visible in the regions where you can reference the identifier without having to qualify it:
• A block can look up to the enclosing block.
• A block cannot look down to enclosed blocks.

Qualify an Identifier
• The qualifier can be the label of an enclosing block.
• Qualify an identifier by using the block label prefix.

```
<<outer>>
DECLARE
birthdateDATE;
BEGIN
DECLARE
birthdateDATE;
BEGIN
...
outer.birthdate:=
TO_DATE('03-AUG-1976',
'DD-MON-YYYY');
END;
....
END;
```

Operators in PL/SQL
• Logical
• Arithmetic
• Concatenation
• Parentheses to control order of operations same as in SQL

• Exponential operator (**)

3. Interacting with the Oracle Server

You can use SELECT statements to populate variables with values queried from a row in a table.
You can use DML commands to modify the data in a database table. However, remember the following points about PL/SQL blocks while using DML statements and transaction control commands in PL/SQL blocks:
•The keyword END signals the end of a PL/SQL block, not the end of a transaction. Just as a block can span multiple transactions, a transaction can span multiple blocks.
•PL/SQL does not directly support data definition language (DDL) statements, such as CREATE TABLE, ALTER TABLE, or DROP TABLE.
•PL/SQL does not support data control language (DCL) statements, such as GRANT or REVOKE.

Retrieve data from the database with a SELECT statement.
Syntax:

```
SELECT  select_list
INTO{variable_name[, variable_name]...
| record_name}
FROM table
[WHERE condition];
```

Guidelines for Retrieving Data in PL/SQL
•Terminate each SQL statement with a semicolon (;).
•The INTO clause is required for the SELECT statement when it is embedded in PL/SQL.
•The WHERE clause is optional and can be used to specify input variables, constants, literals, or PL/SQL expressions.
• Queries must return one and only one row. A query that returns more than one row or no row generates an error.

PL/SQL manages with these errors by raising standard exceptions, which you can trap in the exception section of the block with the NO_DATA_FOUND and TOO_MANY_ROWS exceptions (exception handling is covered in a subsequent lesson). Code SELECT statements to return a single row.

Naming Conventions

```
DECLARE
hire_date      employees.hire_date%TYPE;
sysdatehire_date%TYPE;
employee_id    employees.employee_id%TYPE := 176;
BEGIN
SELECT hire_date,sysdate
INTO   hire_date,sysdate
FROM   employees
WHERE  employee_id = employee_id;
END;
/
```

In potentially ambiguous SQL statements, the names of database columns take precedence over the names of local variables.

Manipulating Data Using PL/SQL
•The INSERT statement adds new rows of data to the table.
•The UPDATE statement modifies existing rows in the table.
•The DELETE statement removes unwanted rows from the table.
•The MERGE statement selects rows from one table to update or insert into another table. The decision whether to update or insert into the target table is based on a condition in the ON clause.

Updating Data
Note: PL/SQL variable assignments always use :=, and SQL column assignments always use =.

Merging Data

```
DECLARE
v_empnoEMPLOYEES.EMPLOYEE_ID%TYPE := 100;
BEGIN
MERGE INTO copy_empc
USING employees e
ON (c.employee_id = v_empno)
WHEN MATCHED THEN
UPDATE SET
c.first_name     = e.first_name,
c.last_name      = e.last_name,
c.email          = e.email,
c.phone_number   = e.phone_number,
c.hire_date      = e.hire_date,
c.job_id         = e.job_id,
c.salary         = e.salary,
c.commission_pct = e.commission_pct,
c.manager_id     = e.manager_id,
c.department_id  = e.department_id
WHEN NOT MATCHED THEN
INSERT VALUES(e.employee_id,  e.first_name, e.last_name,
e.email, e.phone_number, e.hire_date, e.job_id,
e.salary, e.commission_pct, e.manager_id,
e.department_id);
END;
/
```

Naming Conventions

Identifier	Naming_Convention
Identifier	Naming Convention
Variable	v_name
Constant	c_name
Cursor	name_cursor
Exception	e_name
Table type	name_table_type
Table	name_table

Record type	name_record_type
Record	name_record
iSQL*Plus substitution variable (also referred to as substitution parameter)	p_name
iSQL*Plus host or bind variable	g_name

Note: There is no possibility for ambiguity in the SELECT clause because any identifier in the SELECT clause must be a database column name. There is no possibility for ambiguity in the INTO clause because identifiers in the INTO clause must be PL/SQL variables. There is the possibility of confusion only in the WHERE clause.

SQL Cursor
• A cursor is a private SQL work area.
• There are two types of cursors:
– Implicit cursors
– Explicit cursors
• The Oracle server uses implicit cursors to parse and execute your SQL statements.
• Explicit cursors are explicitly declared by the programmer.

SQL Cursor Attributes
Using SQL cursor attributes, you can test the outcome of your SQL statements.
SQL%ROWCOUNT
Number of rows affected by the most recent SQL statement (an integer value)
SQL%FOUND
Boolean attribute that evaluates to TRUE if the most recent SQL statement affects one or more rows
SQL%NOTFOUND
Boolean attribute that evaluates to TRUE if the most recent SQL statement does not affect any rows
SQL%ISOPEN
Always evaluates to FALSE because PL/SQL closes implicit cursors immediately after they are executed

Example:

```
VARIABLE rows_deleted VARCHAR2(30)
DECLARE
v_employee_id employees.employee_id%TYPE := 176;
BEGIN
DELETE FROM  employees
WHERE       employee_id = v_employee_id;
:rows_deleted := (SQL%ROWCOUNT ||
' row deleted.');
END;
/
PRINT rows_deleted
```

Transaction Control Statements
As with Oracle server, DML transactions start at the first command that follows a COMMIT or ROLLBACK, and end on the next successful COMMIT or ROLLBACK.

To mark an intermediate point in the transaction processing, use SAVEPOINT.

4. Writing Control Structures

Controlling PL/SQL Flow of Execution
• Conditional IF statements:
– IF-THEN-END IF
– IF-THEN-ELSE-END IF
– IF-THEN-ELSIF-END IF

Guidelines
•You can perform actions selectively based on conditions that arebeing met.
•When writing code, remember the spelling of the keywords:
– ELSIF is one word.
– END IF is two words.
•If the controlling Boolean condition is TRUE, the associated sequence of statements is executed; if the controlling Boolean condition is FALSE or NULL, the associated sequence of statements is passed over. Any number of ELSIF clauses is permitted.
•Indent the conditionally executed statements for clarity.

Note: Any arithmetic expression containing null values evaluates to null.

CASE Expressions:
Example

```
SET SERVEROUTPUT ON
DEFINE p_grade = a
DECLARE
v_grade CHAR(1) := UPPER('&p_grade');
v_appraisal VARCHAR2(20);
BEGIN
v_appraisal :=
CASE v_grade
WHEN 'A' THEN 'Excellent'
WHEN 'B' THEN 'Very Good'
WHEN 'C' THEN 'Good'
ELSE 'No such grade'
END;
DBMS_OUTPUT.PUT_LINE ('Grade: '|| v_grade || '
Appraisal ' || v_appraisal);
END;
/
```

Handling Nulls
When working with nulls, you can avoid some common mistakes by keeping in mind the following rules:
• Simple comparisons involving nulls always yield NULL.
• Applying the logical operator NOT to a null yields NULL.
• In conditional control statements, if the condition yields NULL, its associated sequence of statements is not executed.

• There are three loop types:
– Basic loop
– FOR loop
– WHILE loop

Basic Loops
Syntax:

```
LOOP            --delimiter
statement1;     --statements
. . .
EXIT [WHEN condition];        --EXIT statement
END LOOP;    --delimiter
```

condition is a Boolean variable or
expression (TRUE, FALSE,
or NULL);

WHILE Loops
Syntax:

```
WHILE condition LOOP
statement1;
statement2;
. . .
END LOOP;
```

Use the WHILE loop to repeat statements while a condition is TRUE. Condition is evaluated at the beginning of each iteration.

Note: If the condition yields NULL, the loop is bypassed and control passes to the next statement.

FOR Loops
Syntax:

```
FOR counter IN [REVERSE]
lower_bound..upper_bound LOOP
statement1;
statement2;
. . .
END LOOP;
```

• Use a FOR loop to shortcut the test for the number of iterations.
• Do not declare the counter; it is declared implicitly.
• 'lower_bound .. upper_bound' is required syntax.

Note: The sequence of statements is executed each time the counter is incremented, as determined by the two bounds. The lower bound and upper bound of the loop range can be literals, variables, or expressions, but must evaluate to integers. The lower bound and upper bound are inclusive in the loop range. If the lower bound of the

loop range evaluates to a larger integer than the upper bound, the sequence of statements will not be executed, provided REVERSE has not been used.
For example the following, statement is executed only once:
FOR i IN 3 .. 3 LOOP statement1; END LOOP;

Guidelines While Using Loops
• Use the basic loop when the statements inside the loop must execute at least once.
• Use the WHILE loop if the condition has to be evaluated at the start of each iteration.
• Use a FOR loop if the number of iterations is known.

Nested Loops and Labels
• Nest loops to multiple levels.
• Use labels to distinguish between blocks and loops.
• Exit the outer loop with the EXIT statement that references the label.

Example
```
BEGIN
<<Outer_loop>>
LOOP
v_counter := v_counter+1;
EXIT WHEN v_counter>10;
<<Inner_loop>>
LOOP
...
EXIT Outer_loop WHEN total_done = 'YES';
--Leave both loops
EXIT WHEN inner_done = 'YES';
--Leave inner loop only
...
END LOOP Inner_loop;
...
END LOOP Outer_loop;
END;
```

5. Working with Composite Data Types

Composite Data Types
Are of two types:
– PL/SQL RECORDs
– PL/SQL Collections
. INDEX BY Table
. Nested Table
. VARRAY

[PL/SQL Records]
A record is a group of related data items stored in fields, each with its own name and data type

• Must contain one or more components of any scalar RECORD, or INDEX BY table data type, called fields

• Are not the same as rows in a database table
• Treat a collection of fields as a logical unit
• Each record defined can have as many fields as necessary.
• Records can be assigned initial values and can be defined as NOT NULL.
• Fields without initial values are initialized to NULL.
• The DEFAULT keyword can also be used when defining fields.
• You can define RECORD types and declare user-defined records in the declarative part of any block, subprogram, or package.
• You can declare and reference nested records. One record can be the component of another record.

Creating a PL/SQL Record :
Syntax

```
TYPE type_name IS RECORD
(field_declaration[, field_declaration]…);
identifiertype_name;
```

Where field_declaration is:

```
field_name {field_type | variable%TYPE
| table.column%TYPE | table%ROWTYPE}
[[NOT NULL] {:= | DEFAULT} expr]
```

Example

```
TYPEemp_record_type IS RECORD
(employee_idNUMBER(6) NOT NULL := 100,
last_name        employees.last_name%TYPE,
job_id           employees.job_id%TYPE);
emp_recordemp_record_type;
```

Fields in a record are accessed by name. To reference or initialize an individual field, use dot notation and the following syntax: record_name.field_name

Declaring Records with the %ROWTYPE Attribute: emp_record employees%ROWTYPE;

A user-defined record and a %ROWTYPE record never have the same data type.

Advantages of Using %ROWTYPE :
• The number and data types of the underlying database columns need not be known.
• The number and data types of the underlying database column may change at run time.
• The attribute is useful when retrieving a row with the SELECT * statement.

Example

```
DEFINE employee_number = 124
DECLARE
emp_recemployees%ROWTYPE;
BEGIN
SELECT * INTOemp_recFROM   employees
WHERE  employee_id = &employee_number;
INSERT INTO retired_emps(empno,ename, job, mgr,hiredate,
```

```
leavedate,sal,comm,deptno)
VALUES (emp_rec.employee_id,emp_rec.last_name,emp_rec.job_id,
emp_rec.manager_id,emp_rec.hire_date, SYSDATE,emp_rec.salary,
emp_rec.commission_pct,emp_rec.department_id);
COMMIT;
END;
```

[INDEX BY Tables]

- Are composed of two components:
  - Primary key of data type BINARY_INTEGER
  - Column of scalar or record data type
- Can increase in size dynamically because they are unconstrained

Creating an INDEX BY Table

Syntax:

```
TYPE type_name IS TABLE OF
{column_type | variable%TYPE
| table.column%TYPE} [NOT NULL]
| table.%ROWTYPE
[INDEX BY BINARY_INTEGER];
identifiertype_name;
```

Example:

```
TYPE ename_table_type IS TABLE OF
employees.last_name%TYPE
INDEX BY BINARY_INTEGER;
ename_table ename_table_type;
```

Referencing an INDEX BY Table

Syntax:

```
INDEX_BY_table_name(primary_key_value)
```

Example:

```
ename_table(3)
```

The magnitude range of a BINARY_INTEGER is -2147483647 ... 2147483647, so the primary key value can be negative. Indexing does not need to start with 1.

Using INDEX BY Table Methods

Method	Description
EXISTS(n)	Returns TRUE if the nth element in a PL/SQL table exists
COUNT	Returns the number of elements that a PL/SQL table currently contains
FIRST, FIRST,LAST	Returns the first and last (smallest and largest) index numbers in a PL/SQL table. Returns NULL if the PL/SQL table is empty.
PRIOR(n)	Returns the index number that precedes index n in a PL/SQL table
NEXT(n)	Returns the index number that succeeds index n in a PL/SQL table
TRIM	TRIM removes one element from the end of a PL/SQL table.
TRIM(n)	Removes n elements from the end of a PL/SQL table.
DELETE	DELETE removes all elements from a PL/SQL table.
DELETE(n)	Removes the nth element from a PL/SQL table.

DELETE(m, n)	Removes all elements in the range m ... n from a PL/SQL table.
EXTEND	Increases size of Collection by 1 or number specified, ie. EXTEND(n)

A INDEX BY table method is a built-in procedure or function that operates on tables and is called using dot notation.
Syntax: table_name.method_name[(parameters)]

[INDEX BY Table of Records]
Example

```
DECLARE
TYPE dept_table_type IS TABLE OF
departments%ROWTYPE
INDEX BY BINARY_INTEGER;
dept_table dept_table_type;
--Each element of dept_table is a record
```

```
SET SERVEROUTPUT ON
DECLARE
TYPE emp_table_type is table of
employees%ROWTYPE INDEX BY BINARY_INTEGER;
my_emp_table emp_table_type;
v_countNUMBER(3):= 104;
BEGIN
FOR i IN 100..v_count
LOOP
SELECT * INTO my_emp_table(i) FROM employees
WHERE employee_id = i;
END LOOP;
FOR i IN my_emp_table.FIRST..my_emp_table.LAST
LOOP
DBMS_OUTPUT.PUT_LINE(my_emp_table(i).last_name);
END LOOP;
END;
```

6. Writing Explicit Cursors

Cursor Types
Implicit: Implicit cursors are declared by PL/SQL implicitly for all DML and PL/SQL SELECT statements, including queries that return only one row.
Explicit: For queries that return more than one row, explicit cursors are declared and named by the programmer and manipulated through specific statements in the block's executable actions.

Syntax:

```
CURSOR cursor_name IS
select_statement;
```

• Do not include the INTO clause in the cursor declaration.

• If processing rows in a specific sequence is required, use the ORDER BY clause in the query.

Opening the Cursor
Syntax: OPEN cursor_name;

• Open the cursor to execute the query and identify the active set.
• If the query returns no rows, no exception is raised.
• Use cursor attributes to test the outcome after a fetch.

OPEN is an executable statement that performs the following operations:
1. Dynamically allocates memory for a context area that eventually contains crucial processing information.
2. Parses the SELECT statement.
3. Binds the input variables—sets the value for the input variables by obtaining their memory addresses
4. Identifies the active set—the set of rows that satisfy the search criteria. Rows in the active set are not retrieved into variables when the OPEN statement is executed. Rather, the FETCH statement retrieves the rows.
5. Positions the pointer just before the first row in the active set.

Fetching Data from the Cursor
Syntax: FETCH cursor_name INTO[variable1, variable2, ...] | record_name];

The FETCH statement performs the following operations:
1. Reads the data for the current row into the output PL/SQL variables.
2. Advances the pointer to the next row in the identified set.

Closing the Cursor
Syntax: CLOSE cursor_name;

CLOSE Statement
The CLOSE statement disables the cursor, and the active set becomes undefined. Close the cursor after completing the processing of the SELECT statement. This step allows the cursor to be reopened, if required. Therefore, you can establish an active set several times. There is a maximum limit to the number of open cursors per user, which is determined by the OPEN_CURSORS parameter in the database parameter file.
OPEN_CURSORS = 50 by default.

Example

```
OPENemp_cursor
FOR i IN 1..10 LOOP
FETCHemp_cursor INTO v_empno, v_ename;
...
END LOOP;
CLOSEemp_cursor;
END;
```

Explicit Cursor Attributes
Obtain status information about a cursor.

Attribute	Type	Description
%ISOPEN	Boolean	Evaluates to TRUE if the cursor is open
%NOTFOUND	Boolean	Evaluates to TRUE if the most recent fetch does not return a row
%FOUND	Boolean	Evaluates to TRUE if the most recent fetch returns a row; complement of %NOTFOUND
%ROWCOUNT	Number	Evaluates to the total number of rows returned so far

The %ISOPEN Attribute
• You can fetch rows only when the cursor is open. Use the %ISOPEN cursor attribute to determine whether the cursor is open.

The %NOTFOUND and %ROWCOUNT Attributes

```
LOOP
FETCH c1 INTO my_ename, my_sal, my_hiredate;
EXIT WHEN c1%NOTFOUND;
...
END LOOP;
```

Before the first fetch, %NOTFOUND evaluates to NULL. So, if FETCH never executes successfully, the loop is never exited. That is because the EXIT WHEN statement executes only if its WHEN condition is true. To be safe, use the following EXIT statement instead:

```
EXIT WHEN c1%NOTFOUND OR c1%NOTFOUND IS NULL;
```

If a cursor is not open, referencing it with %NOTFOUND raises INVALID_CURSOR.

%ROWCOUNT
When its cursor or cursor variable is opened, %ROWCOUNT is zeroed. Before the first fetch, %ROWCOUNT yields 0. Thereafter, it yields the number of rows fetched so far. The number is incremented if the last fetch returned a row. In the next example, you use %ROWCOUNT to take action if more than ten rows have been fetched:

```
LOOP
FETCH c1 INTO my_ename, my_deptno;

IF c1%ROWCOUNT > 10 THEN
...
END IF;
...
END LOOP;
```

If a cursor is not open, referencing it with %ROWCOUNT raises INVALID_CURSOR. the row count is not incremented if the fetch does not retrieve any rows.

Cursors and Records:

```
DECLARE
CURSOREmp_cursor IS
SELECT employee_id, last_name
FROM   employees;
emp_recordemp_cursor%ROWTYPE;
BEGIN
```

```
OPENemp_cursor;
LOOP
FETCHemp_cursor INTOemp_record;
EXIT WHENemp_cursor%NOTFOUND;
INSERT INTO temp_list (empid,empname)
VALUES (emp_record.employee_id,emp_record.last_name);
END LOOP;
COMMIT;
CLOSEemp_cursor;
END;
/
```

Cursor FOR Loops

```
DECLARE
CURSOREmp_cursor IS
SELECT last_name, department_id
FROM   employees;
BEGIN
FOREmp_record INEmp_cursor LOOP
--implicit open and implicit fetch occur
IFemp_record.department_id = 80 THEN
...
END LOOP; --implicit close occurs
END;
```

Cursor FOR Loops Using Subqueries

No need to declare the cursor.
Example:

```
BEGIN
FOREmp_record IN (SELECT last_name, department_id
FROM   employees) LOOP
--implicit open and implicit fetch occur
IFemp_record.department_id = 80 THEN
...
END LOOP; --implicit close occurs
END;
```

7. Advanced Explicit Cursor Concepts

Cursors with Parameters
Syntax:

```
CURSOR cursor_name
[(parameter_namedatatype, ...)]
IS
select_statement;
```

OPEN cursor_name(parameter_value,.....) ;

• Pass parameter values to a cursor when the cursor is opened and the query is executed.
• Open an explicit cursor several times with a different active set each time.

The following statements open the cursor and returns different active sets:
OPENemp_cursor(60, v_emp_job);
OPENemp_cursor(90, 'AD_VP');

You can pass parameters to the cursor used in a cursor FOR loop:
BEGIN
FOR emp_record IN emp_cursor(50, 'ST_CLERK') LOOP ...

The FOR UPDATE Clause
Syntax:
SELECT...
FROM...
FOR UPDATE [OF column_reference][NOWAIT];

• Explicit locking allows you to deny access for the duration of a transaction.
• Lock the rows before the update or delete.

Example
Retrieve the employees who work in department 80 and update their salary.

```
DECLARE
CURSOREmp_cursor IS
SELECT employee_id, last_name, department_name
FROM   employees,departments
WHERE  employees.department_id =
departments.department_id
AND employees.department_id = 80
FOR UPDATE OF salary NOWAIT;
```

The FOR UPDATE clause identifies the rows that will be updated or deleted, then locks each row in the result set. This is useful when you want to base an update on the existing values in a row.
In that case, you must make sure the row is not changed by another user before the update.

It is not mandatory that the FOR UPDATE OF clause refer to a column, but it is recommended for better readability and maintenance.

The optional NOWAIT keyword tells Oracle not to wait if requested rows have been locked by another user. Control is immediately returned to your program so that it can do other work before trying again to acquire the lock. If you omit the NOWAIT keyword, Oracle waits until the rows are available.

If you have a large table, you can achieve better performance by using the LOCK TABLE statement to lock all rows in the table. However, when using LOCK TABLE, you cannot use the WHERE CURRENT OF clause and must use the notation WHERE column = identifier.

The WHERE CURRENT OF Clause
Syntax:
WHERE CURRENT OF cursor ;

• Use cursors to update or delete the current row.
• Include the FOR UPDATE clause in the cursor query to lock the rows first.
• Use the WHERE CURRENT OF clause to reference the current row from an explicit cursor.

This allows you to apply updates and deletes to the row currently being addressed, without the need to explicitly reference the ROWID.

Example

The WHERE CURRENT OF Clause

```
DECLARE
CURSORsal_cursor IS
SELECT e.department_id, employee_id, last_name, salary
FROM   employees e, departments d
WHERE  d.department_id = e.department_id
and   d.department_id = 60
FOR UPDATE OF salary NOWAIT;
BEGIN
FORemp_record INsal_cursor
LOOP
IFemp_record.salary < 5000 THEN
UPDATE employees
SET    salary =emp_record.salary * 1.10
WHERE CURRENT OFsal_cursor;
END IF;
END LOOP;
END;
/
```

Cursors with Subqueries
Example:

```
DECLARE
CURSOR my_cursor IS
SELECT t1.department_id, t1.department_name,
t2.staff
FROM   departments t1, (SELECT department_id,
COUNT(*) AS STAFF
FROM employees
GROUP BY department_id) t2
WHERE t1.department_id = t2.department_id
AND   t2.staff >= 3;
```

...

8. Handling Exceptions

Two Methods for Raising an Exception
•An Oracle error occurs and the associated exception is raised automatically. For example, if the error ORA-01403 occurs when no rows are retrieved from the database in a SELECT statement, then PL/SQL raises the exception NO_DATA_FOUND.
•You raise an exception explicitly by issuing the RAISE statement within the block. The exception being raised may be either user-defined or predefined.

Exception Types
Implicitly raised
• Predefined Oracle Server
• Non predefined Oracle Server
Explicitly raised
• User-defined

Trapping Exceptions
Syntax:
EXCEPTION
WHEN exception1 [OR exception2 . . .] THEN
statement1;
statement2;
. . .
[WHEN exception3 [OR exception4 . . .] THEN
statement1;
statement2;
. . .]
[WHEN OTHERS THEN
statement1;
statement2;
. . .]

WHEN OTHERS Exception Handler
The OTHERS handler traps all exceptions not already trapped. Some Oracle tools have their own predefined exceptions that you can raise to cause events in the application. The OTHERS handler also traps these exceptions.

Trapping Exceptions Guidelines
• The EXCEPTION keyword starts exception-handling section.
• Several exception handlers are allowed.
• Only one handler is processed before leaving the block.
• WHEN OTHERS is the last clause.
•You can have only one OTHERS clause.
•Exceptions cannot appear in assignment statements or SQL statements.

• Sample predefined exceptions:

– NO_DATA_FOUND
– TOO_MANY_ROWS
– INVALID_CURSOR
– ZERO_DIVIDE
– DUP_VAL_ON_INDEX

Trapping non predefined Oracle Server Errors
In PL/SQL, the PRAGMA EXCEPTION_INIT tells the compiler to associate an exception name with an Oracle error number. That allows you to refer to any internal exception by name and to write a specific handler for it.

Note: PRAGMA (also called pseudo instructions) is the keyword that signifies that the statement is a compiler directive, which is not processed when the PL/SQL block is executed. Rather, it directs the PL/SQL compiler to interpret all occurrences of the exception name within the block as the associated Oracle server error number.

Non pre defined Error
Example
Trap for Oracle server error number –2292, an integrity constraint violation.

```
DEFINE p_deptno= 10
DECLARE
BEGIN
DELETE FROM departments
WHERE department_id = &p_deptno;
COMMIT;
EXCEPTION
WHEN e_emps_remaining   THEN
DBMS_OUTPUT.PUT_LINE (’Cannot remove dept ’ ||
TO_CHAR(&p_deptno) || ’.  Employees exist. ’);
END;
```

non predefined Error
Trap for Oracle server error number –2292, an integrity constraint violation.

```
PRAGMA EXCEPTION_INIT
(e_emps_remaining, -2292);
e_emps_remaining
e_emps_remaining EXCEPTION;
```

Functions for Trapping Exceptions
• SQLCODE: Returns the numeric value for the error code
• SQLERRM: Returns the message associated with the error number

SQLCODE	*Value Description*
0	No exception encountered
1	User-defined exception
+100	NO_DATA_FOUND exception
negative number	Another Oracle server error number

Example:

```
DECLARE
v_error_code      NUMBER;
v_error_message   VARCHAR2(255);
BEGIN
...
EXCEPTION
...
WHEN OTHERS THEN
ROLLBACK;
v_error_code := SQLCODE ;
v_error_message := SQLERRM ;
INSERT INTO errors
VALUES(v_error_code, v_error_message);
```

User-Defined Exceptions

Example:

```
DEFINE p_department_desc= 'Information Technology '
DEFINE P_department_number = 300

DECLARE
e_invalid_department EXCEPTION;
BEGIN
UPDATEdepartments
SETdepartment_name = '&p_department_desc'
WHEREdepartment_id = &p_department_number;
IF SQL%NOTFOUND THEN
RAISE e_invalid_department;
END IF;
COMMIT;
EXCEPTION
WHEN e_invalid_department  THEN
DBMS_OUTPUT.PUT_LINE('No such department id.');
END;
```

Propagating an Exception in a Sub block

When a sub block handles an exception, it terminates normally, and control resumes in the enclosing block immediately after the sub block END statement. However, if PL/SQL raises an exception and the current block does not have a handler for that exception, the exception propagates in successive enclosing blocks until it finds a handler. If none of these blocks handle the exception, an unhandled exception in the host environment results. When the exception propagates to an enclosing block, the remaining executable actions in that block are bypassed.

The RAISE_APPLICATION_ERROR
Procedure
Syntax:

```
raise_application_error (error_number,
```

message[, {TRUE | FALSE}]);

• You can use this procedure to issue user-defined error messages from stored subprograms.
• You can report errors to your application and avoid returning unhandled exceptions.

error_number is a user-specified number for the exception between –20000 and –20999.
message is the user-specified message for the exception. It is a character string up to 2,048 bytes long.
TRUE | FALSE is an optional Boolean parameter (If TRUE, the error is placed on the stack of previous errors. If FALSE, the default, the error replaces all previous errors.)

9. Creating Procedures

What Is a Procedure?
• A procedure is a type of subprogram that performs an action.
• A procedure can be stored in the database, as a schema object, for repeated execution.

Syntax for Creating Procedures

```
CREATE [OR REPLACE] PROCEDURE procedure_name
[(parameter1 [mode1] datatype1,
parameter2 [mode2] datatype2,
. . .)]
IS|AS
PL/SQL Block;
```

• The REPLACE option indicates that if the procedure exists, it will be dropped and replaced with the new version created by the statement.
• PL/SQL block starts with either BEGIN or the declaration of local variables and ends with either END or END procedure_name.

Formal Versus Actual Parameters
• Formal parameters: variables declared in the parameter list of a subprogram specification
Example:

```
CREATE PROCEDURE raise_sal(p_idNUMBER, p_amountNUMBER)
...
END raise_sal;
```

• Actual parameters: variables or expressions referenced in the parameter list of a subprogram call
Example:

```
raise_sal(v_id, 2000)
```

Type of Parameter Description
IN (default) passes a constant value from the calling environment into the procedure
OUT Passes a value from the procedure to the calling environment
IN OUT Passes a value from the calling environment into the procedure and a possibly different value from the procedure back to the calling environment using the same parameter

A formal parameter of IN mode cannot be assigned a value. That is, an IN parameter cannot be modified in the

body of the procedure. An OUT or INOUT parameter must be assigned a value before returning to the calling environment.IN parameters can be assigned a default value in the parameter list. OUT and INOUT parameters cannot be assigned default values. By default, the IN parameter is passed by reference and the OUT and INOUT parameters are passed by value. To improve performance with OUT and INOUT parameters, the compiler hint NOCOPY can be used to request to pass by reference.

IN parameters are passed as constants from the calling environment into the procedure. Attempts to change the value of an IN parameter result in an error.

The name of the script file need not be the same as the procedure name. (The script file is on the client side and the procedure is being stored on the database schema.)

IN Parameters: Example

```
CREATE OR REPLACE PROCEDURE raise_salary
(p_id IN employees.employee_id%TYPE)
IS
BEGIN
UPDATE employees
SET    salary = salary * 1.10
WHERE  employee_id = p_id;
END raise_salary;
```

OUT Parameters: Example

```
emp_query.sql
CREATE OR REPLACE PROCEDURE query_emp
(p_id      IN   employees.employee_id%TYPE,
p_name    OUT  employees.last_name%TYPE,
p_salary  OUT  employees.salary%TYPE,
p_comm    OUT  employees.commission_pct%TYPE)
IS
BEGIN
SELECT   last_name, salary, commission_pct
INTO     p_name, p_salary, p_comm
FROM     employees
WHERE    employee_id = p_id;
END query_emp;
/
```

Viewing OUT Parameters

• Load and run the emp_query.sql script file to create the QUERY_EMP procedure.
• Declare host variables, execute the QUERY_EMPprocedure, and print the value of the global variable G_NAME.

```
VARIABLE g_nameVARCHAR2(25)
VARIABLE g_salNUMBER
VARIABLE g_commNUMBER
EXECUTE query_emp(171, :g_name, :g_sal, :g_comm)
PRINT g_name
```

INOUT Parameters

Calling environment	FORMAT_PHONE procedure
p_phone_no '8006330575'	'(800)633-0575'

```
CREATE OR REPLACE PROCEDURE format_phone
(p_phone_no IN OUT VARCHAR2)
IS
BEGIN
p_phone_no := '(' || SUBSTR(p_phone_no,1,3) ||
')' || SUBSTR(p_phone_no,4,3) ||
'-' || SUBSTR(p_phone_no,7);
END format_phone;
```

Viewing IN OUT Parameters

```
VARIABLE g_phone_no VARCHAR2(15)
BEGIN
:g_phone_no := '8006330575';
END;
/
PRINT g_phone_no
EXECUTE format_phone (:g_phone_no)
PRINT g_phone_no
```

Parameter Passing Methods

Method	*Description*
Positional	Lists values in the order in which the parameters are declared
Named association	Lists values in arbitrary order by associating each one with its parameter name, using special syntax (=>)
Combination	Lists the first values positionally, and the remainder using the special syntax of the named method

DEFAULT Option for Parameters

```
CREATE OR REPLACE PROCEDURE add_dept
(p_name  IN departments.department_name%TYPE
DEFAULT 'unknown',
p_loc   IN departments.location_id%TYPE
DEFAULT 1700)
IS
BEGIN
INSERT INTO departments(department_id,
department_name, location_id)
VALUES (departments_seq.NEXTVAL, p_name, p_loc);
END add_dept;
/
```

You can assign default values only to parameters of the IN mode. OUT and INOUT parameters are not permitted to have default values. If default values are passed to these types of parameters, you get the following compilation error:
PLS-00230: OUT and IN OUT formal parameters may not have default expressions.

Examples of Passing Parameters

```
BEGIN
add_dept;
add_dept ('TRAINING', 2500);
add_dept ( p_loc => 2400, p_name =>'EDUCATION');
add_dept ( p_loc => 1200) ;
END;
/
SELECT department_id, department_name, location_id
FROM departments;
```

Declaring Subprograms

```
leave_emp2.sql
CREATE OR REPLACE PROCEDURE leave_emp2
(p_id  IN  employees.employee_id%TYPE)
IS
PROCEDURE log_exec
IS
BEGIN
INSERT INTO log_table (user_id, log_date)
VALUES (USER, SYSDATE);
END log_exec;
BEGIN
DELETE FROM employees
WHERE employee_id = p_id;
log_exec;
END leave_emp2;
/
```

Subprograms declared in this manner are called local subprograms (or local modules). Because they are defined within a declaration section of another program, the scope of local subprograms is limited to the parent (enclosing) block in which they are defined.

You must declare the subprogram in the declaration section of the block, and it must be the last item, after all the other program items. For example, a variable declared after the end of the subprogram, before the BEGIN of the procedure, will cause a compilation error.

Invoking a Procedure from an Anonymous PL/SQL Block

```
DECLARE
```

```
v_id NUMBER := 163;
BEGIN
raise_salary(v_id);    --invoke procedure
COMMIT;
...
END;
```

Invoking a Procedure from Another Procedure

process_emps.sql

```
CREATE OR REPLACE PROCEDURE process_emps
IS
CURSOR emp_cursor IS
SELECT employee_id
FROM   employees;
BEGIN
FOR emp_rec IN emp_cursor
LOOP
raise_salary(emp_rec.employee_id);
END LOOP;
COMMIT;
END process_emps;
/
```

How Handled Exceptions Affect the Calling Procedure

Any data manipulation language (DML) statements issued before the exception was raised remain as part of the transaction.

Handled Exceptions

```
CREATE PROCEDURE p2_ins_dept(p_locid NUMBER) IS
v_did NUMBER(4);
BEGIN
DBMS_OUTPUT.PUT_LINE('Procedure p2_ins_dept started');
INSERT INTO departments VALUES (5, 'Dept 5', 145, p_locid);
SELECT department_id INTO v_did FROM employees
WHERE employee_id = 999;
END;

CREATE PROCEDURE p1_ins_loc(p_lid NUMBER, p_city VARCHAR2)
IS
v_city VARCHAR2(30); v_dname VARCHAR2(30);
BEGIN
DBMS_OUTPUT.PUT_LINE('Main Procedure p1_ins_loc');
INSERT INTO locations (location_id, city) VALUES (p_lid, p_city);
SELECT city INTO v_city FROM locations WHERE location_id = p_lid;
DBMS_OUTPUT.PUT_LINE('Inserted city '||v_city);
DBMS_OUTPUT.PUT_LINE('Invoking the procedure p2_ins_dept ...');
p2_ins_dept(p_lid);
```

```
EXCEPTION
WHEN NO_DATA_FOUND THEN
DBMS_OUTPUT.PUT_LINE('No such dept/loc for any employee');
END;
```

p2_ins_dept is called in p1_ins_loc The SELECT in p2_ins_dept returns NO_DATA_FOUND exception & is not handled in the procedure. Hence control goes to next statement of calling procedure where the exception is handled in this block.

As the exception is handled, the DML in the P2_INS_DEPT procedure is not rolled back and is part of the transaction of the P1_INS_LOC procedure.

How Unhandled Exceptions Affect the Calling Procedure?
The P2_NOEXCEP procedure has a SELECT statement that selects DEPARTMENT_ID for a non existing employee and raises a NO_DATA_FOUND exception. Because this exception is not handled in the P2_NOEXCEP procedure, the control returns to the calling procedure P1_NOEXCEP. The exception is not handled. Because the exception is not handled, the DML in the P2_NOEXCEP procedure is rolled back along with the transaction of the P1_NOEXCEP procedure.

Removing Procedures
Drop a procedure stored in the database.
Syntax:
DROP PROCEDURE procedure_name

Example: DROP PROCEDURE raise_salary;

Benefits of Subprograms
Procedures and functions have many benefits in addition to modularizing application development:
•Easy maintenance: Subprograms are located in one location and hence it is easy to:
–Modify routines online without interfering with other users
–Modify one routine to affect multiple applications
–Modify one routine to eliminate duplicate testing
•Improved data security and integrity
–Controls indirect access to database objects from non privileged users with security privileges. As a subprogram is executed with its definer's right by default, it is easy to restrict the access privilege by granting a privilege only to execute the subprogram to a user.
–Ensures that related actions are performed together, or not at all, by funneling activity for related tables through a single path
•Improved performance
–After a subprogram is compiled, the parsed code is available in the shared SQL area of the server and subsequent calls to the subprogram use this parsed code. This avoids reparsing for multiple users.
–Avoids PL/SQL parsing at run time by parsing at compile time
–Reduces the number of calls to the database and decreases network traffic by bundling commands
•Improves code clarity: Using appropriate identifier names to describe the action of the routines reduces the need for comments and enhances the clarity of the code.

10. Creating Functions

Syntax for Creating Functions

```
CREATE [OR REPLACE] FUNCTION function_name
[(parameter1 [mode1] datatype1,
parameter2 [mode2] datatype2,
. . .)]
RETURN datatype
IS|AS
PL/SQL Block;
```

Creating a Stored Function by Using iSQL*Plus:

Example

get_salary.sql

```
CREATE OR REPLACE FUNCTION get_sal
(p_id  IN employees.employee_id%TYPE)
RETURN NUMBER
IS
v_salary employees.salary%TYPE :=0;
BEGIN
SELECT salary
INTO   v_salary
FROM   employees
WHERE  employee_id = p_id;
RETURN v_salary;
END get_sal;
/
```

There can be a RETURN statement in the exception section of the program also.

In a function, there must be at least one execution path that leads to a RETURN statement. Otherwise, you get a Function returned without value error at run time.

Invoking Functions in SQL Expressions:

Example

```
CREATE OR REPLACE FUNCTION tax(p_value IN NUMBER)
RETURN NUMBER IS
BEGIN
RETURN (p_value * 0.08);
END tax;
/
SELECT employee_id, last_name, salary, tax(salary)
FROM   employees
WHERE  department_id = 100;
```

Locations to Call User-Defined Functions

- Select list of a SELECT command
- Condition of the WHERE and HAVING clauses
- CONNECT BY, START WITH, ORDER BY, and GROUP BY clauses

• VALUES clause of the INSERT command
• SET clause of the UPDATE command

Restrictions on Calling Functions from
SQL Expressions
To be callable from SQL expressions, a user-defined function must:
• Be a stored function
• Accept only IN parameters
• Accept only valid SQL data types, not PL/SQL specific types, as parameters
• Return data types that are valid SQL data types, not PL/SQL specific types
•Parameters to a PL/SQL function called from a SQL statement must use positional notation. Named notation is not supported.
•Stored PL/SQL functions cannot be called from the CHECK constraint clause of a CREATE or ALTERTABLE command or be used to specify a default value for a column.
• Functions called from SQL expressions cannot contain DML statements.
• Functions called from UPDATE/DELETE statements on a table T cannot contain DML on the same table T.
• Functions called from a DML statement on a table T cannot query the same table.
• Functions called from SQL statements cannot contain statements that end the transactions.
• Calls to subprograms that break the previous restriction are not allowed in the function.

Note: Only stored functions are callable from SQL statements. Stored procedures cannot be called. The ability to use a user-defined PL/SQL function in a SQL expression is available with PL/SQL 2.1 and later. Tools using earlier versions of PL/SQL do not support this functionality. Prior to Oracle9i, user-defined functions can be only single-row functions. Starting with Oracle9i, user-defined functions can also be defined as aggregate functions.

Note: Functions those are callable from SQL expressions cannot contain OUT and INOUT parameters.
Other functions can contain parameters with these modes, but it is not recommended.

Restrictions on Calling from SQL

```
CREATE OR REPLACE FUNCTIONdml_call_sql(p_salNUMBER)
RETURN NUMBER IS
BEGIN
INSERT INTO employees(employee_id, last_name, email,
hire_date, job_id, salary)
VALUES(1, 'employee 1', 'emp1@company.com',
SYSDATE, 'SA_MAN', 1000);
RETURN (p_sal+ 100);
END;
/
UPDATE employees SET salary =dml_call_sql(2000)
WHERE employee_id = 170;
```

Removing Functions
Drop a stored function.
DROP FUNCTION function_name
Syntax:
Example:
DROP FUNCTION get_sal;

• All the privileges granted on a function are revoked when the function is dropped.
• The CREATE OR REPLACE syntax is equivalent to dropping a function and recreating it. Privileges granted on the function remain the same when this syntax is used.

Comparing Procedures and Functions
Procedure
-Execute as a PL/SQL statement
-No RETURN clause in the header
-Can return none, one, or many values
-Can contain a RETURN statement

Function
-Invoke as part of an expression
-Must contain a RETURN clause in the header
-Must return a single value
-Must contain at least one RETURN statement

A procedure containing one OUT parameter can be rewritten as a function containing a RETURN statement.

11. Managing Subprograms

Required Privileges
DBA grants
CREATE (ANY) PROCEDURE
ALTER ANY PROCEDURE System privileges
DROP ANY PROCEDURE
EXECUTE ANY PROCEDURE

Owner grants
EXECUTE Object privileges

System and Object Privileges
There are more than 80 system privileges. Privileges that use the word CREATE or ANY are system privileges; for example, GRANT ALTER ANY TABLE TO green;. System privileges are assigned by user SYSTEM or SYS.
Object privileges are rights assigned to a specific object within a schema and always include the name of the object. For example, Scott can assign privileges to Green to alter his EMPLOYEES table as follows:
GRANT ALTER ON employees TO green; to create a PL/SQL subprogram, you must have the system privilege CREATEPROCEDURE. You can alter, drop, or execute PL/SQL subprogram without any further privileges being required. If a PL/SQL subprogram refers to any objects that are not in the same schema, you must be granted access to these explicitly, not through a role. If the ANY keyword is used, you can create, alter, drop, or execute your own subprograms and those in another schema. Note that the ANY keyword is optional only for the CREATE PROCEDURE privilege. You must have the EXECUTE object privilege to invoke the PL/SQL subprogram if you are not the owner and do not have the EXECUTEANY system privilege. By default the PL/SQL subprogram executes under the security domain of the owner.

Note: The keyword PROCEDURE is used for stored procedures, functions, and packages.

Direct Access
•From the PERSONNEL schema, provide object privileges on the EMPLOYEES table to Scott.
•Scott creates the QUERY_EMP procedure that queries the EMPLOYEES table.

Indirect Access
Scott provides the EXECUTE object privilege to Green on the QUERY_EMP procedure. By default the PL/SQL subprogram executes under the security domain of the owner. This is referred to as definer's-rights. Because Scott has direct privileges to EMPLOYEES and has created a procedure called QUERY_EMP, Green can retrieve information from the EMPLOYEES table by using the QUERY_EMP procedure.

Using Invoker's-Rights
The procedure executes with the privileges of the user.

Example

```
CREATE PROCEDURE query_employee
(p_id IN employees.employee_id%TYPE,
p_name OUT employees.last_name%TYPE,
p_salary OUT employees.salary%TYPE,
p_commOUT
employees.commission_pct%TYPE)
        AUTHID CURRENT_USER
IS
BEGIN
SELECT last_name, salary,
commission_pct
INTO  p_name, p_salary, p_comm
FROM  employees
WHERE employee_id=p_id;
END query_employee;
```

Invoker's-Rights
To ensure that the procedure executes using the security of the executing user, and not the owner, use AUTHID CURRENT_USER. This ensures that the procedure executes with the privileges and schema context of its current user. If you wanted to explicitly state that the procedure should execute using the owner's privileges, then use AUTHID DEFINER.

Stored Information	Description	Access Method
General	Object information	The USER_OBJECTS data dictionary view
Source code	Text of the procedure	The USER_SOURCE data dictionary view
Parameters	Mode: IN/ OUT/ IN OUT, data type	iSQL*Plus: DESCRIBE command
P-code	Compiled object code	Not accessible
Compile errors	PL/SQL syntax errors	The USER_ERRORS data dictionary view, iSQL*Plus: SHOW ERRORS command

Run-time debug information	User-specified debug variables and messages	The DBMS_OUTPUT Oracle-supplied package

List All Procedures and Functions
SELECT object_name, object_type
FROM user_objects
WHERE object_type in ('PROCEDURE',
'FUNCTION')ORDER BY object_name;

List the Code of Procedures and Functions
SELECT text
FROM user_source
WHERE name = 'QUERY_EMPLOYEE'
ORDER BY line;

List Compilation Errors by Using USER_ERRORS
SELECT line || '/' || position POS, text
FROM user_errors
WHERE name = 'LOG_EXECUTION'
ORDER BY line;

List Compilation Errors by Using SHOW ERRORS
SHOW ERRORS PROCEDURE log_execution

SHOWERRORS
Use SHOWERRORS without any arguments at the SQL prompt to obtain compilation errors for the last object you compiled. You can also use the command with a specific program unit. The syntax is as follows:
SHOW ERRORS [{FUNCTION|PROCEDURE|PACKAGE|PACKAGE
BODY|TRIGGER|VIEW} [schema.]name]

Describing Procedures and Functions
To display a procedure or function and its parameter list, use the iSQL*Plus DESCRIBE command.

Debugging PL/SQL Program Units
• The DBMS_OUTPUT package:
– Accumulates information into a buffer
– Allows retrieval of the information from the buffer
• Autonomous procedure calls (for example, writing the output to a log table)
• Software that uses DBMS_DEBUG
– Procedure Builder
– Third-party debugging software

12. Creating Packages

Packages:
• Group logically related PL/SQL types, items, and subprograms

• Consist of two parts:
– Specification
– Body
• Cannot be invoked, parameterized, or nested
• Allow the Oracle server to read multiple objects into memory at once

A package usually has a specification and a body, stored separately in the database. The specification is the interface to your applications. It declares the types, variables, constants, exceptions, cursors, and subprograms available for use. The package specification may also include PRAGRMAs, which are directives to the compiler.

Scope of the Construct: Public Description: Can be referenced from any Oracle server environment
Placement within the Package: Declared within the package specification and may be defined within the package body

Scope of the Construct: Private
Description: Can be referenced only by other constructs which are part of the same package Placement within the Package: Declared and defined within the package body

Developing a Package
• Saving the text of the CREATE PACKAGE statement in two different SQL files facilitates later modifications to the package.
• A package specification can exist without a package body, but a package body cannot exist without a package specification.

To create packages, you declare all public constructs within the package specification.
•Specify the REPLACE option when the package specification already exists.
•Initialize a variable with a constant value or formula within the declaration, if required; otherwise, the variable is initialized implicitly to NULL.

Syntax:
CREATE [OR REPLACE] PACKAGE package_name
IS|AS
public type and item declarations
subprogram specifications
END package_name;

Creating a Package Body:
Example
comm_pack.sql

```
CREATE OR REPLACE PACKAGE BODY comm_package
IS
FUNCTION  validate_comm (p_comm IN NUMBER)
RETURN BOOLEAN
IS
v_max_comm    NUMBER;
BEGIN
SELECT    MAX(commission_pct)
```

```
INTO     v_max_comm
FROM     employees;
IF   p_comm > v_max_comm THEN RETURN(FALSE);
ELSE   RETURN(TRUE);
END IF;
END validate_comm;

PROCEDURE  reset_comm (p_comm   IN  NUMBER)
IS
BEGIN
IF  validate_comm(p_comm)
THEN   g_comm:=p_comm;  --reset global variable
ELSE
RAISE_APPLICATION_ERROR(-20210,'Invalid commission');
END IF;
END reset_comm;
END comm_package;
```

Invoking Package Constructs :
After the package is stored in the database, you can invoke a package construct within the package or from outside the package, depending on whether the construct is private or public. When you invoke a package procedure or function from within the same package, you do not need to qualify its name.

Example 2: Invoke a package procedure from iSQL*Plus.

```
EXECUTE comm_package.reset_comm(0.15)
```

Example 3: Invoke a package procedure in a different schema.

```
EXECUTE scott.comm_package.reset_comm(0.15)
```

Example 4: Invoke a package procedure in a remote database.

```
EXECUTE comm_package.reset_comm@ny(0.15)
```

Declaring a Bodiless Package

```
CREATE OR REPLACE PACKAGE global_consts IS
mile_2_kilo    CONSTANT  NUMBER  :=  1.6093;
kilo_2_mile    CONSTANT  NUMBER  :=  0.6214;
yard_2_meter   CONSTANT  NUMBER  :=  0.9144;
meter_2_yard   CONSTANT  NUMBER  :=  1.0936;
END global_consts;
/
EXECUTE DBMS_OUTPUT.PUT_LINE('20 miles = '||20*
global_consts.mile_2_kilo||' km')
```

Referencing a Public Variable from a Stand-Alone Procedure
Example:

```
CREATE OR REPLACE PROCEDURE meter_to_yard
(p_meter IN NUMBER, p_yard OUT NUMBER)
IS
BEGIN
p_yard := p_meter * global_consts.meter_2_yard;
```

```
END meter_to_yard;
/
VARIABLE yard NUMBER
EXECUTE  meter_to_yard (1, :yard)
```

Removing Packages:
To remove the package specification and the body, use the following syntax:
DROP PACKAGE package_name;
To remove the package body, use the following syntax :
DROP PACKAGE BODY package_name;

- Changes to the package specification require recompilation of each referencing subprogram.

Command	*Task*
CREATE [OR REPLACE] PACKAGE	Create (or modify) an existing package specification
CREATE [OR REPLACE] PACKAGE BODY	Create (or modify) an existing package body
DROP PACKAGE	Remove both the package specification and the package body
DROP PACKAGE BODY	Remove the package body only

13. More Package Concepts

Overloading
• Enables you to use the same name for different subprograms inside a PL/SQL block, a subprogram, or a package
• Requires the formal parameters of the subprograms to differ in number, order, or data type family
• Enables you to build more flexibility because a user or application is not restricted by the specific data type or number of formal parameters Note: Only local or packaged subprograms can be overloaded. You cannot overload stand-alone subprograms.

Restrictions
You cannot overload:
•Two subprograms if their formal parameters differ only in data type and the different datatypes are in the same family (NUMBER and DECIMAL belong to the same family)
•Two subprograms if their formal parameters differ only in subtype and the different subtypes are based on types in the same family (VARCHAR and STRING are PL/SQL subtypes of VARCHAR2)
•Two functions that differ only in return type, even if the types are in different families. You get a run-time error when you overload subprograms with the above features.
Note: The above restrictions apply if the names of the parameters are also the same. If you use different names for the parameters, then you can invoke the subprograms by using named notation for the parameters.

Resolving Calls
The compiler tries to find a declaration that matches the call. It searches first in the current scope and then, if necessary, in successive enclosing scopes. The compiler stops searching if it finds one or more subprogram declarations in which the name matches the name of the called subprogram. For like-named subprograms at the same level of scope, the compiler needs an exact match in number, order, and data type between the actual and formal parameters.

Overloading: Example

over_pack.sql

```
CREATE OR REPLACE PACKAGE over_pack
IS
PROCEDURE add_dept
(p_deptnoIN departments.department_id%TYPE,
p_name IN departments.department_name%TYPE
DEFAULT 'unknown',
p_loc IN departments.location_id%TYPE DEFAULT 0);
PROCEDURE add_dept
(p_name IN departments.department_name%TYPE
DEFAULT 'unknown',
p_loc  IN departments.location_id%TYPE DEFAULT 0);
END over_pack;
/
```

over_pack_body.sql

```
CREATE OR REPLACE PACKAGE BODY over_pack  IS
PROCEDURE add_dept
(p_deptnoIN departments.department_id%TYPE,
p_name IN departments.department_name%TYPE DEFAULT 'unknown',
p_loc  IN departments.location_id%TYPE DEFAULT 0)
IS
BEGIN
INSERT INTO departments (department_id,
department_name, location_id)
VALUES  (p_deptno, p_name, p_loc);
END add_dept;
PROCEDURE add_dept
(p_name IN departments.department_name%TYPE DEFAULT 'unknown',
p_loc  IN departments.location_id%TYPE DEFAULT 0)
IS
BEGIN
INSERT INTO departments (department_id,
department_name, location_id)
VALUES  (departments_seq.NEXTVAL, p_name, p_loc);
END add_dept;
END over_pack;
/
```

If you call ADD_DEPT with an explicitly provided department ID, PL/SQL uses the first version of the procedure. If you call ADD_DEPT with no department ID, PL/SQL uses the second version.

```
EXECUTE over_pack.add_dept (980,'Education',2500)
EXECUTE over_pack.add_dept ('Training', 2400)
```

- Most built-in functions are overloaded.
- For example, see the TO_CHAR function of the STANDARD package.

```
FUNCTION TO_CHAR (p1 DATE) RETURN VARCHAR2;
FUNCTION TO_CHAR (p2 NUMBER) RETURN VARCHAR2;
```

FUNCTION TO_CHAR (p1 DATE, P2 VARCHAR2) RETURN VARCHAR2;
FUNCTION TO_CHAR (p1 NUMBER, P2 VARCHAR2) RETURN VARCHAR2;
• If you redeclare a built-in subprogram in a PL/SQL program, your local declaration overrides the global declaration.

To be able to access the built-in subprogram, you need to qualify it with its package name. For example, if you redeclare the TO_CHAR function, to access the built-in function you refer it as: STANDARD.TO_CHAR.
If you redeclare a built-in subprogram as a stand-alone subprogram, to be able to access your subprogram you need to qualify it with your schema name, for example, SCOTT.TO_CHAR.

Using Forward Declarations
PL/SQL does not allow forward references. You must declare an identifier before using it.
Therefore, a subprogram must be declared before calling it.

PL/SQL enables for a special subprogram declaration called a forward declaration. It consists of the subprogram specification terminated by a semicolon. You can use forward declarations to do the following:
•Define subprograms in logical or alphabetical order
•Define mutually recursive subprograms
•Group subprograms in a package
Mutually recursive programs are programs that call each other directly or indirectly.

Using Forward Declarations

```
CREATE OR REPLACE PACKAGE BODY forward_pack
IS
PROCEDURE calc_rating(. . .);   --forward declaration
PROCEDURE award_bonus(. . .)
IS--subprograms defined
BEGIN                --in alphabetical order
calc_rating(. . .);
. . .
END;
PROCEDURE calc_rating(. . .)
IS
BEGIN
. . .
END;
END forward_pack;
/
```

•The formal parameter list must appear in both the forward declaration and the subprogram body.
•The subprogram body can appear anywhere after the forward declaration, but both must appear in the same program unit.

Creating a One-Time-Only Procedure

```
CREATE OR REPLACE PACKAGE taxes
IS
tax   NUMBER;
...  --declare all public procedures/functions
```

```
END  taxes;
/
CREATE OR REPLACE PACKAGE BODY taxes
IS
... --declare all private variables
... --define public/private procedures/functions
BEGIN
SELECT   rate_value
INTO     tax
FROM     tax_rates
WHERE    rate_name = 'TAX';
END taxes;
/
```

A one-time-only procedure is executed only once, when the package is first invoked within the user session. In the preceding slide, the current value for TAX is set to the value in the TAX_RATES table the first time the TAXES package is referenced.
Note: Initialize public or private variables with an automatic, one-time-only procedure when the derivation is too complex to embed within the variable declaration. In this case, do not initialize the variable in the declaration, because the value is reset by the one-time-only procedure. The keyword END is not used at the end of a one-time-only procedure. Observe that in the example in the slide, there is no END at the end of the one-time-only procedure.

Restrictions on Package Functions Used in SQL
For the Oracle server to execute a SQL statement that calls a stored function, it must know the purity level of a stored functions, that is, whether the functions are free of side effects.
Following restrictions apply to stored functions called from SQL expressions:
•A function called from a query or DML statement may not end the current transaction, create or rollback to a savepoint, or alter the system or session
•A function called from a query statement or from a parallelized DML statement may not execute a DML statement or otherwise modify the database
•A function called from a DML statement may not read or modify the particular table being modified by that DML statement

```
CREATE OR REPLACE PACKAGE taxes_pack
IS
FUNCTION tax (p_value IN NUMBER) RETURN NUMBER;
END taxes_pack;
/
User Defined Package: taxes_pack
CREATE OR REPLACE PACKAGE BODY taxes_pack
IS
FUNCTION tax (p_value IN NUMBER) RETURN NUMBER
IS
v_rate NUMBER := 0.08;
BEGIN
RETURN (p_value * v_rate);
END tax;
```

```
END taxes_pack;
/
```

Invoking a User-Defined Package Function from a SQL Statement

```
SELECT taxes_pack.tax(salary), salary, last_name
FROM   employees;
```

Calling Package Functions

You call PL/SQL functions the same way that you call built-in SQL functions.

Example

Call the TAX function (in the TAXES_PACK package) from a SELECT statement.

Note: If you are using Oracle versions prior to 8i, you need to assert the purity level of the function in the package specification by using PRAGMA RESTRICT_REFERENCES. If this is not specified, you get an error message saying that the function TAX does not guarantee that it will not update the database while invoking the package function in a query.

Controlling the Persistent State of a Package Variable

You can keep track of the state of a package variable or cursor, which persists throughout the user session, from the time the user first references the variable or cursor to the time the user disconnects.

1. Initialize the variable within its declaration or within an automatic, one-time-only procedure.
2. Change the value of the variable by means of package procedures.
3. The value of the variable is released when the user disconnects.

Controlling the Persistent State of a Package Cursor

Example:

```
CREATE OR REPLACE PACKAGE pack_cur
IS
CURSOR c1 IS  SELECT employee_id
FROM   employees
ORDER BY employee_id DESC;
PROCEDURE proc1_3rows;
PROCEDURE proc4_6rows;
END pack_cur;
/
```

Use the following steps to control a public cursor:

1. Declare the public (global) cursor in the package specification.
2. Open the cursor and fetch successive rows from the cursor, using one (public) packaged procedure PROC1_3ROWS.
3. Continue to fetch successive rows from the cursor, and then close the cursor by using another (public) packaged procedure, PROC4_6ROWS.

```
CREATE OR REPLACE PACKAGE BODY pack_cur  IS
v_empnoNUMBER;
PROCEDURE proc1_3rows IS
BEGIN
OPEN c1;
LOOP
```

```
FETCH c1 INTO v_empno;
DBMS_OUTPUT.PUT_LINE('Id :' ||(v_empno));
EXIT WHEN c1%ROWCOUNT >= 3;
END LOOP
END proc1_3rows;
PROCEDURE proc4_6rows IS
BEGIN
LOOP
FETCH c1 INTO v_empno;
DBMS_OUTPUT.PUT_LINE('Id :' ||(v_empno));
EXIT WHEN c1%ROWCOUNT >= 6;
END LOOP;
CLOSE c1;
END proc4_6rows;
END pack_cur;
/
```

1. Open the cursor and fetch successive rows from the cursor by using one packaged procedure, PROC1_3ROWS.
2. Continue to fetch successive rows from the cursor and close the cursor, using another packaged procedure, PROC4_6ROWS.

```
SET SERVEROUTPUT ON
EXECUTE pack_cur.proc1_3rows
EXECUTE pack_cur.proc4_6rows
```

The state of a package variable or cursor persists across transactions within a session. The state does not persist from session to session for the same user, nor does it persist from user to user.

PL/SQL Tables and Records in Packages:

```
CREATE OR REPLACE PACKAGE BODYemp_package IS
PROCEDURE read_emp_table
(p_emp_table OUTemp_table_type) IS
i BINARY_INTEGER := 0;
BEGIN
FOREmp_record IN (SELECT * FROM employees)
LOOP
emp_table(i) :=emp_record;
i:= i+1;
END LOOP;
END read_emp_table;
ENDemp_package;
/

CREATE OR REPLACE PACKAGEemp_package IS
TYPEemp_table_type IS TABLE OF employees%ROWTYPE
INDEX BY BINARY_INTEGER;
```

```
PROCEDURE read_emp_table
(p_emp_table OUTemp_table_type);
END emp_package;
/
```

Passing Tables of Records to Procedures or Functions inside a Package
Invoke the READ_EMP_TABLE procedure from an anonymous PL/SQL block, using iSQL*Plus.

```
DECLARE
v_emp_table emp_package.emp_table_type;
BEGIN
emp_package.read_emp_table(v_emp_table);
DBMS_OUTPUT.PUT_LINE('An example: '||v_emp_table(4).last_name);
END;
/
```

To invoke the procedure READ_EMP_TABLE from another procedure or any PL/SQL block outside the package, the actual parameter referring to the OUT parameter P_EMP_TABLE must be prefixed with its package name. In the example above, the variable V_EMP_TABLE is declared of the EMP_TABLE_TYPE type with the package name added as a prefix.

14. Oracle Supplied Packages

Components of the DBMS_SQL Package
The DBMS_SQL package uses dynamic SQL to access the database.

Function / Procedure	Description
OPEN_CURSOR	Opens a new cursor and assigns a cursor ID number
PARSE	Parses the DDL or DML statement: that is, checks the statement's syntax and associates it with the opened cursor (DDL executed when parsed)statements are immediately
BIND_VARIABLE	Binds the given value to the variable identified by its name in the parsed statement in the given cursor
EXECUTE	Executes the SQL statement and returns the number of rows processed
FETCH_ROWS	Retrieves a row for the specified cursor (for multiple rows,call in a loop)
CLOSE_CURSOR	Closes the specified cursor

Using DBMS_SQL
Use dynamic SQL to delete rows

```
VARIABLE deleted NUMBER
EXECUTE  delete_all_rows('employees', :deleted)
PRINT deleted
CREATE OR REPLACE PROCEDURE delete_all_rows
(p_tab_name IN VARCHAR2, p_rows_del OUT NUMBER)
IS
cursor_name   INTEGER;
BEGIN
cursor_name := DBMS_SQL.OPEN_CURSOR;
```

```
DBMS_SQL.PARSE(cursor_name, 'DELETE FROM '||p_tab_nam
DBMS_SQL.NATIVE );
p_rows_del := DBMS_SQL.EXECUTE (cursor_name);
DBMS_SQL.CLOSE_CURSOR(cursor_name);
END;
/
```

How to Process Dynamic DML

1. Use OPEN_CURSOR to establish an area in memory to process a SQL statement.
2. Use PARSE to establish the validity of the SQL statement.
3. Use the EXECUTE function to run the SQL statement. This function returns the number of row processed.
4. Use CLOSE_CURSOR to close the cursor.

Use the EXECUTE IMMEDIATE for native dynamic SQL with better performance.

• INTO is used for single-row queries and specifies the variables or records into which column values are retrieved.

• USING is used to hold all bind arguments. The default parameter mode is IN.

Using the EXECUTE IMMEDIATE statement

```
EXECUTE IMMEDIATE dynamic_string
[INTO {define_variable
[, define_variable] ... | record}]
[USING [IN|OUT|IN OUT] bind_argument
[, [IN|OUT|IN OUT] bind_argument] ... ];
```

Dynamic SQL Using EXECUTE IMMEDIATE

```
CREATE PROCEDURE del_rows
(p_table_name  IN  VARCHAR2,
p_rows_deldOUT NUMBER)
IS
BEGIN
EXECUTE IMMEDIATE 'delete from '||p_table_name;
p_rows_deld:= SQL%ROWCOUNT;
END;
/
```

```
VARIABLE deleted NUMBER
EXECUTE del_rows('test_employees',:deleted)
PRINT deleted
```

Using the DBMS_DDL Package

The DBMS_DDL Package:

• Provides access to some SQL DDL statements from stored procedures

• Includes some procedures:

– ALTER_COMPILE (object_type, owner, object_name)

```
DBMS_DDL.ALTER_COMPILE('PROCEDURE','A_USER','QUERY_EMP')
```

– ANALYZE_OBJECT (object_type, owner, name, method)
DBMS_DDL.ANALYZE_OBJECT('TABLE','A_USER','JOBS','COMPUTE')
Note: This package runs with the privileges of calling user, rather than the package owner SYS.

Practical Uses
•You can recompile your modified PL/SQL program units by using DBMS_DDL.ALTER_COMPILE. The object type must be either procedure, function, package, package body, or trigger.
•You can analyze a single object, using DBMS_DDL.ANALYZE_OBJECT. (There is a way of analyzing more than one object at a time, using DBMS_UTILITY.) The object type should be TABLE, CLUSTER, or INDEX. The method must be COMPUTE, ESTIMATE, or DELETE.
•This package gives developers access to ALTER and ANALYZE SQL statements through PL/SQL environments.

15. Manipulating Large Objects

What Is a LOB?
LOBs are used to store large unstructured data such as text, graphic images, films, and sound waveforms.

There are two categories of LOBs:
•Internal LOBs (CLOB, NCLOB, and BLOB) are stored in the database.
•External files (BFILE) are stored outside the database.

•BLOB represents a binary large object, such as a video clip.
•CLOB represents a character large object.
•NCLOB represents a multi byte character large object.
•BFILE represents a binary file stored in an operating system binary file outside the database. The BFILE column or attribute stores a file locator that points to the external file.

The Oracle9i Server performs implicit conversion between CLOB and VARCHAR2 data types. The other implicit conversions between LOBs are not possible. BFILE scan be accessed only in read-only mode from an Oracle server.

Contrasting LONG and LOB Data Types

LONG and LONG RAW	**LOB**
Single LONG column per table	Multiple LOB columns per table
Up to 2 GB	Up to 4 GB
SELECT returns data	Sequential access to data
Data stored in-line	Data stored in-line or out-of-line
SELECT returns locator	Random access to data
Cannot be partitioned	Can be partitioned
Cannot be attributes of a user-defined object type	Can be attributes of a user-defined object type
Tables with long column cannot be replicated	Tables with LOB columns can be replicated

---

1. Collections

A collection can be loosely defined as a group of ordered elements, all of the same type,that allows programmatic

access to its elements through an index

--Define a PL/SQL record type representing a book:

```
TYPE book_rec IS RECORD
   (title                  book.title%TYPE,
    author                 book.author_last_name%TYPE,
    year_published published_date.%TYPE));
```

--define a PL/SQL table containing entries of type book_rec:

```
Type book_rec_tab IS TABLE OF book_rec%TYPE
    INDEX BY BINARY_INTEGER;

my_book_rec  book_rec%TYPE;
my_book_rec_tab book_rec_tab%TYPE;
...
...
my_book_rec := my_book_rec_tab(5);
find_authors_books(my_book_rec.author);
...
...
```

2. Varrays

```
DECLARE
    TYPE genres IS VARRAY(4) OF book_genre.genre_name%TYPE;
    Fiction_genres genres;
BEGIN
    fiction_genres := genres('MYSTERY','SUSPENSE', 'ROMANCE','HORROR');
END;

--Add a new genre.
IF adding_new_genre  THEN

     --Is this genre id already in the collection?
     IF NOT fiction_genres.EXISTS(v_genre_id)     THEN
       --**Add** another element to the varray.
       fiction_genres.EXTENDS(1);
       fiction_genres(v_genre_id) := v_genre;
   END IF;
    --Display the total # of elements.
    DBMS_OUTPUT.PUT_LINE('Total # of entries in fiction_genres is :
                         '||fiction_genres.COUNT();

END IF;
...
...
--Remove all entries.
IF deleting_all_genres THEN
     Fiction_genres.DELETE();
```

```
END IF;
```

The advantage that Varrays (and Nested Tables) have over Associative Arrays (INDEX BY TABLE) is their ability to be added to the database. For example, you could add the genres type, a Varray, to a DML statement on the library table.

```
CREATE TYPE genres IS VARRAY(4) OF book_genre.genre_name%TYPE;
/
CREATE TABLE book_library (
    library_id    NUMBER,
    name          VARCHAR2(30),
    book_genres   genres);
/

--Insert a new collection into the column on our book_library table.
INSERT INTO book_library (library_id, name, book_genres)
  VALUES (book_library_seq.NEXTVAL,'Brand New Library',
          Genres('FICTION','NON-FICTION', 'HISTORY',
                 'BUSINESS AND FINANCE'));
```

The query SELECT name, book_genres from book_library returns us:

NAME	BOOK_GENRES
Brand New Library	GENRES('FICTION', 'NON-FICTION', 'HISTORY', 'BUSINESS AND FINANCE')

3. Nested Table

```
CREATE TYPE genres_tab IS TABLE OF book_genre.genre_name%TYPE;
/

CREATE TABLE book_library (
    library_id     NUMBER,
    name           VARCHAR2(30),
    book_genres_tab genres_tab)
    NESTED TABLE book_genres_tab STORE AS genres_table;
/


--Insert a record into book_library, with a Nested Table of book genres.
INSERT INTO book_library (library_id, name, book_genres_tab)
  VALUES (book_library_seq.NEXTVAL,'Brand New Library',
          genres_tab('FICTION','NON-FICTION', 'HISTORY', 'BUSINESS AND FINANCE'));
/

--Declare a nested table type
DECLARE
  updated_genres_tab genres_tab;
BEGIN
  updated_genres_tab :=
    genres_tab('FICTION','NON-FICTION','HISTORY','BUSINESS AND FINANCE',
               'SCIENCE','PERIODICALS','MULTIMEDIA');
```

```
  --Update the existing record with a new genres Nested Table.
  UPDATE book_library
    SET book_genres_tab = updated_genres_tab;

END

--1.)Select all genres from library 'Brand New Library' that are like
     '%FICTION%'.
SELECT column_value FROM TABLE(SELECT book_genres_tab
                               FROM book_library
                               WHERE name = 'Brand New Library')
   WHERE column_value LIKE '%FICTION%';
/

COLUMN_VALUE
------------------------------
FICTION
NON-FICTION

--2.)Update entry 'MULTIMEDIA' to a new value.  Only possible with a nested table!!
UPDATE TABLE(SELECT book_genres_tab
             FROM book_library
             WHERE name = 'Brand New Library')
  SET column_value   = 'MUSIC AND FILM'
  WHERE column_value = 'MULTIMEDIA';


--3.)Select all book_genre_tab entries for this library.
SELECT column_value FROM TABLE(SELECT book_genres_tab
                               FROM book_library
                               WHERE name = 'Brand New Library');

COLUMN_VALUE
----------------------------
FICTION
NON-FICTION
HISTORY
BUSINESS AND FINANCE
SCIENCE
PERIODICALS
MULTIMEDIA
```

4. Associative Array (index-by table)

```
TYPE book_title_tab IS TABLE OF book.title%TYPE
    INDEX BY BINARY_INTEGER;
book_titles   book_title_tab

SELECT title FROM book;
```

```
TITLE
-----------------------------
A Farewell to Arms
For Whom the Bell Tolls
The Sun Also Rises
```

```
 FOR cursor_column IN  book_titles.FIRST..book_titles.LAST LOOP
       IF book_titles(cursor_column) = 'A Farewell to Arms' THEN
          book_titles.DELETE(cursor_column);
       END IF;
   END LOOP;
```

With Associative Arrays, it is now possible to index by the title of the book. In fact, there are numerous different indexing options, including by VARCHAR2, using the %TYPE keyword, and more. This is a improvement over indexing everything by an integer then having to shuffle through entries to find what you're looking for. Now, if we want to remove the book A Farewell to Arms, we can use an Associative Array:

```
DECLARE
   TYPE book_title_tab IS TABLE OF book.title%TYPE
       INDEX BY book.title%TYPE;
   book_titles book_title_tab;
BEGIN
      book_titles.DELETE('A Farewell to Arms');
END;
```

- Collection Capabilities -

Has Ability To	*Varray*	*Nested Table*	*Associative Array*
be indexed by non-integer	No	No	Yes
preserve element order	Yes	No	No
be stored in database	Yes	Yes	No
have elements selected indidually in database	Yes	Yes	--
have elements updated indidually in database	Yes	No	--

Varray
Use to preserve ordered list
Use when working with a fixed set, with a known number of entries
Use when you need to store in the database and operate on the Collection as a whole

Nested Table
Use when working with an unbounded list that needs to increase dynamically
Use when you need to store in the database and operate on elements individually

Associative Array
Use when there is no need to store the Collection in the database. Its speed and indexing flexibility make it ideal

for internal application use.

Dimension

```
CREATE DIMENSION [ schema. ]dimension
  level_clause
  [ level_clause ]...
  { hierarchy_clause
  | attribute_clause
  | extended_attribute_clause
  }
    [ hierarchy_clause
    | attribute_clause
    | extended_attribute_clause
    ]... ;
```

Example=>

```
CREATE DIMENSION customers_dim
   LEVEL customer   IS (customers.cust_id)
   LEVEL city       IS (customers.cust_city)
   LEVEL state      IS (customers.cust_state_province)
   LEVEL country    IS (countries.country_id)
   LEVEL subregion  IS (countries.country_subregion)
   LEVEL region     IS (countries.country_region)
   HIERARCHY geog_rollup (
      customer      CHILD OF
      city          CHILD OF
      state         CHILD OF
      country       CHILD OF
      subregion     CHILD OF
      region
   JOIN KEY (customers.country_id) REFERENCES country
   )
   ATTRIBUTE customer DETERMINES
   (cust_first_name, cust_last_name, cust_gender,
    cust_marital_status, cust_year_of_birth,
    cust_income_level, cust_credit_limit)
   ATTRIBUTE country DETERMINES (countries.country_name)
;
```

Example of sql injection

SQL injection occurs when user input is not filtered for escape characters and is then passed into a SQL statement.

These results in the potential manipulation of the statements performed on the database by the end user of the application.

The following line of code illustrates this vulnerability:

```
statement = "SELECT * FROM users WHERE name = '" + userName + "';"
```

This SQL code is designed to pull up the records of the specified username from its table of users. However, if the "userName" variable is crafted in a specific way by a malicious user, the SQL statement may do more than the code author intended. For example, setting the "userName" variable as ' or '1'='1 Or using comments to even block the rest of the query:' or '1'='1';/*' renders this SQL statement by the parent language:

```
SELECT * FROM users WHERE name = '' OR '1'='1';
```

If this code were to be used in an authentication procedure then this example could be used to force the selection of a valid username because the evaluation of '1'='1' is always true.

The following value of "userName" in the statement below would cause the deletion of the "users" table as well as the selection of all data from the "userinfo" table (in essence revealing the information of every user), using an API that allows multiple statements:

```
a';DROP TABLE users; SELECT * FROM userinfo WHERE 't' = 't
```

This input renders the final SQL statement as follows:

```
SELECT * FROM users WHERE name = 'a';DROP TABLE users; SELECT * FROM userinfo WHERE 't' = 't';
```