Database Systems

Introduction to PL/SQL

What is PL/SQL?

- Procedural programming language
  - Uses detailed instructions
  - Processes statements sequentially
- Combines SQL commands with procedural instructions
- Used to perform sequential processing using an Oracle database

PL/SQL Variables

- Variable names must follow the Oracle naming standard
  - Can use reserved words (BEGIN, NUMBER) and table names for variable names, but is not a good practice
- Make variable names descriptive
- Use lower-case letters, and separate words with underscores
  - Example: `current_s_id`

Declaring PL/SQL Variables

- PL/SQL is a strongly-typed language
  - All variables must be declared prior to use
- Syntax for declaring a variable:

```
variable_name data_type_declaration;
```

- Example:

```
current_s_id NUMBER(6);
```

PL/SQL Data Types

- Scalar
  - References a single value
- Composite
  - References a data structure
- Reference
  - References a specific database item
- LOB
  - References a large binary object

Scalar Data Types

- Database scalar data types:
  - VARCHAR2
  - CHAR
  - DATE
  - LONG
  - NUMBER
- Non-database scalar data types:
  - Integers: BINARY_INTEGER, INTEGER, INT, SMALLINT
  - Decimal numbers: DEC, DECIMAL, DOUBLE, PRECISION, NUMERIC, REAL
  - BOOLEAN

Composite Data Types

- Reference multiple data elements, such as a record
- Types:
  - RECORD
  - TABLE
  - VARRAY
    - Tabular structure that can expand or contract as needed

Reference Data Types

- Reference a database item
- Assume data type of item
  - %TYPE: assumes data type of field
  - %ROWTYPE: assumes data type of entire row

PL/SQL Program Structure

```
DECLARE
  Variable declarations
BEGIN
  Program statements
EXCEPTION
  Error-handling statements
END;
```

Variable Declarations

Body

Exception Section

PL/SQL Program Lines

- May span multiple text editor lines
- Each line ends with a semicolon
- Text is not case sensitive

Comment Statements

- Block of comments are delimited with /* */

```
/* <comment that spans more than one
  line of code> */
```

- Single comment line starts with 2 hyphens

```
-- comment on a single line
```

Arithmetic Operators

**	Exponentiation	2 ** 3	8
*	Multiplication	2 * 3	6
/	Division	9/2	4.5
+	Addition	3 + 2	5
-	Subtraction	3 – 2	1
-	Negation	-5	Negative 5

Assignment Statements

- Assignment operator: **:=**
- Variable being assigned to a new value is on left side of assignment operator
- New value is on right side of operator

```
student_name := ‘John Miller’;
student_name := current_student;
```

Displaying PL/SQL Output in SQL*Plus

- Command to activate memory buffer in SQL*Plus to enable output from PL/SQL programs:

```
SQL> SET SERVEROUTPUT ON SIZE buffer_size;

SQL> SET SERVEROUTPUT ON SIZE 4000;
```

Displaying PL/SQL Program Output in SQL*Plus

- Command to output data from a PL/SQL program in SQL*Plus:

```
DBMS_OUTPUT.PUT_LINE('output string');

DBMS_OUTPUT.PUT_LINE('Current Output:');
```

Executing a PL/SQL Program in SQL*Plus

- Copy program code from Notepad to SQL Plus
- Type / to execute

PL/SQL Selection Structures

- IF/END IF:

```
IF condition THEN
  program statements
END IF;
```

- IF/ELSE/END IF:

```
IF condition THEN
  program statements
ELSE
  alternate program statements
END IF;
```

PL/SQL Selection Structures

- IF/ELSIF:

```
IF condition1 THEN
  program statements;
ELSIF condition2 THEN
  alternate program statements;
ELSIF condition3 THEN
  alternate program statements;
. . .
ELSE
  alternate program statements;
END IF;
```

PL/SQL Comparison Operators

Operator	Description	Example
=	Equal	Count = 5
<>, !=	Not Equal	Count <> 5
>	Greater Than	Count > 5
<	Less Than	Count < 5
>=	Greater Than or Equal To	Count >= 5
<=	Less Than or Equal To	Count <= 5

Evaluating NULL Conditions in IF/THEN Structures

- If a condition evaluates as NULL, then it is FALSE
- How can a condition evaluate as NULL?
  - It uses a BOOLEAN variable that has not been initialized
  - It uses any other variable that has not been initialized

Using SQL Commands in PL/SQL Programs

SQL Command Category	Purpose	Examples	Can be used in PL/SQL
Data Definition Language (DDL)	Change database structure	CREATE, ALTER, GRANT, REVOKE	No
Data Manipulation Language (DML)	View or change data	SELECT, INSERT, UPDATE, DELETE	Yes
Transaction Control	Create logical transactions	COMMIT, ROLLBACK	Yes

PL/SQL Loops

- Loop: repeats one or more program statements multiple times until an exit condition is reached
  - Pretest loop: exit condition is tested <u>before</u> program statements are executed
  - Posttest loop: exit condition is tested <u>after</u> program statements are executed

LOOP ... EXIT Loop

Pretest
OR
Posttest

LOOP … EXIT WHEN Loop

```
LOOP
   program statements
   EXIT WHEN condition;
END LOOP;
```

Posttest

WHILE Loop

```
WHILE condition
LOOP
   program statements
END LOOP;
```

Pretest

Numeric FOR Loop

```
FOR counter_variable
IN start_value .. end_value
LOOP
   program statements
END LOOP;
```

Preset number of iterations

Sequential Control

- *GOTO <code block name>*
- The entry point into such a block of code is marked by using the tags<<*userdefinedname*>>

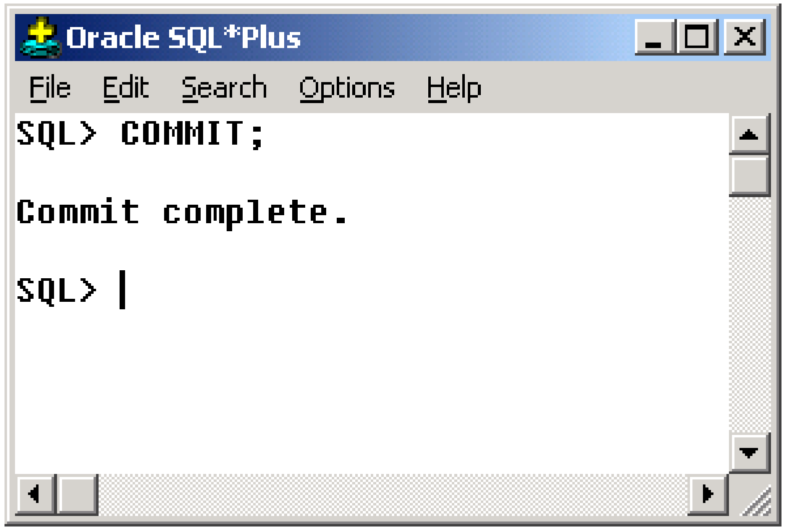
Questions...

- Write a PL/SQL code block to calculate the area of a circle for a value of radius varying from 3 to 7. Store the radius and corresponding values of calculated area in an empty table named **Areas**, consisting of two columns **Radius** and **Area.**
- Write a PL/SQL block of code for inverting a number 5639 to 9365.
- Write a PL/SQL block of code for finding the factorial of a a given number.
- Write a PL/SQL code block that will accept an account number from the user and debit an amount of Rs.2000 from the account if the account has a minimum balance of 500 after the amount is debited. The process is fired on the Accounts table.

Transactions and Security

Transactions

- Oracle starts a transaction with the first SQL statement after a COMMIT, ROLLBACK or connection to the database.
- Oracle ends a transaction with a COMMIT, ROLLBACK or disconnection from the database.



```
SQL> COMMIT;

Commit complete.

SQL> |
```

Transactions

- Until changes are committed (made permanent)
  - You can see the changes when the table is queried but,
  - Others cannot see them when they query query your tables and,
  - You can roll them back (discard them) if you change your mind or need to correct a mistake

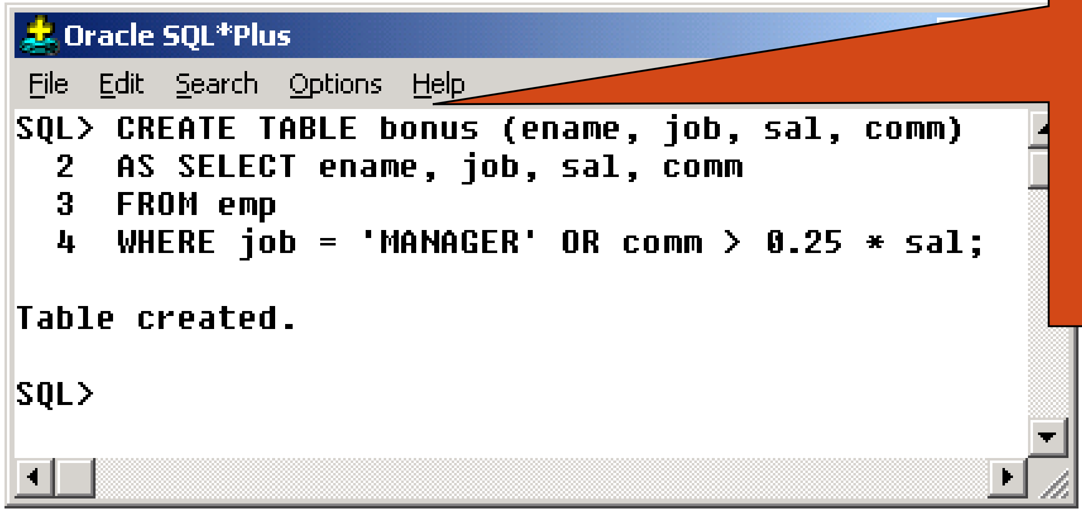
Transactions

- Oracle issues an implicit COMMIT before and after any DDL (Data Definition Language - CREATE, ALTER, DROP) SQL statement.

Transactions

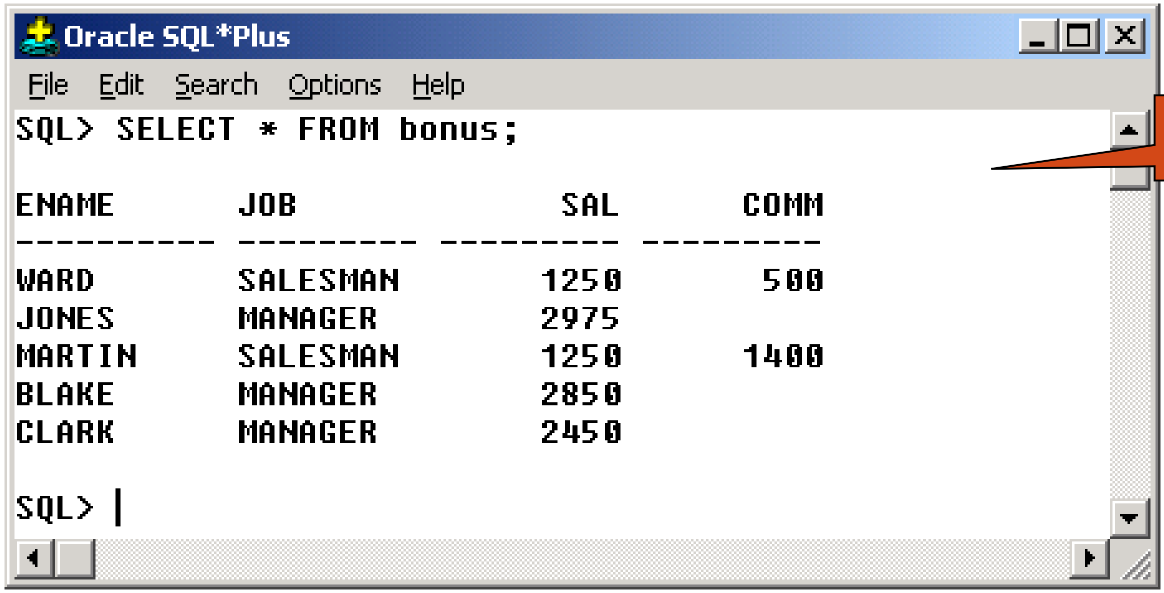
- There are a number of commands to help manage transactions

  - SAVEPOINT savepoint
  - ROLLBACK [TO [SAVEPOINT] savepoint]
  - COMMIT



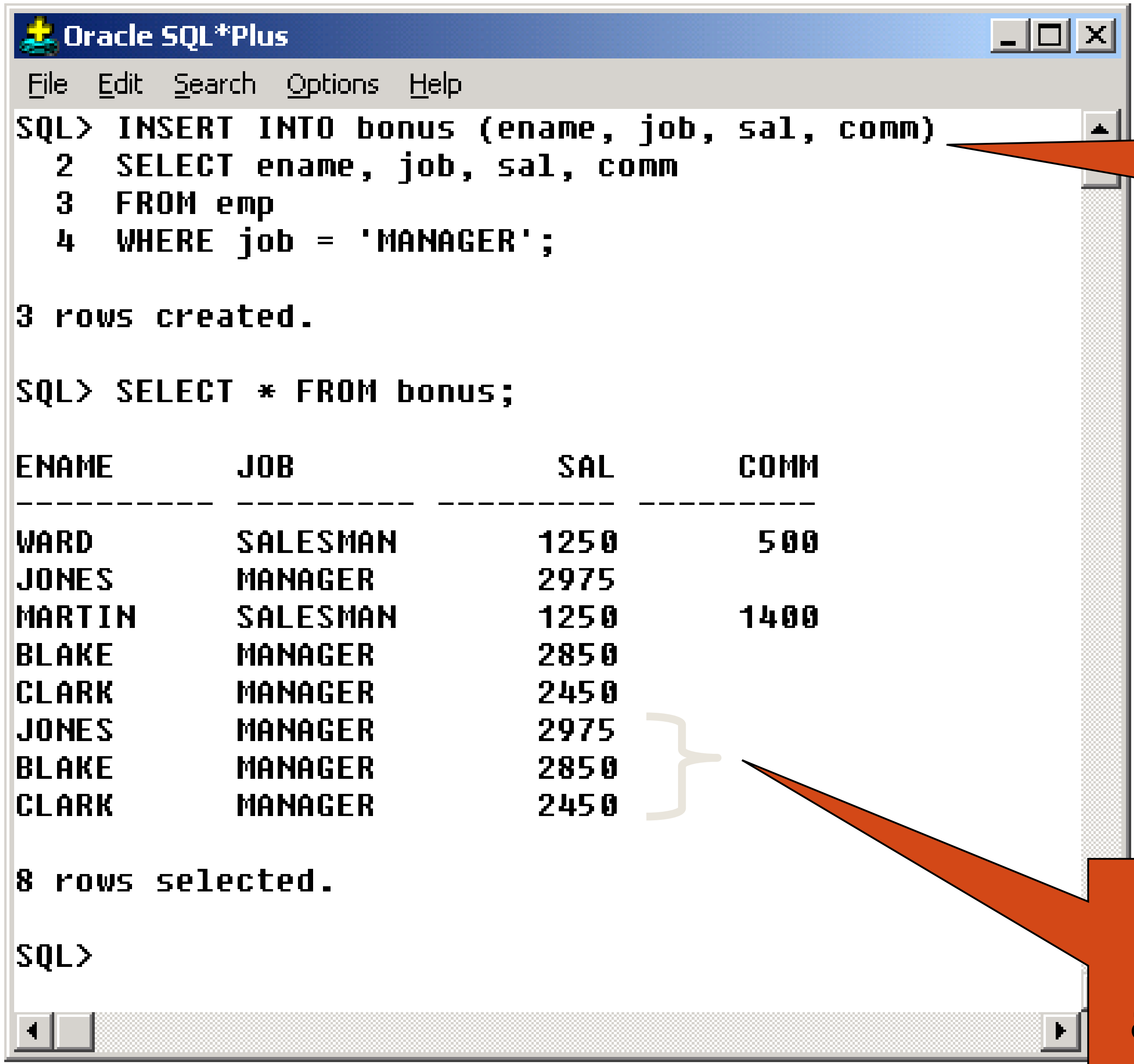
```
SQL> CREATE TABLE bonus (ename, job, sal, comm)
  2  AS SELECT ename, job, sal, comm
  3  FROM emp
  4  WHERE job = 'MANAGER' OR comm > 0.25 * sal;

Table created.

SQL>
```

Creating a table called 'BONUS'
(note the way the table has been created and had rows inserted in one operation)



```
SQL> SELECT * FROM bonus;

ENAME      JOB              SAL       COMM
---------- --------- ---------- ----------
WARD       SALESMAN        1250        500
JONES      MANAGER         2975
MARTIN     SALESMAN        1250       1400
BLAKE      MANAGER         2850
CLARK      MANAGER         2450

SQL> |
```

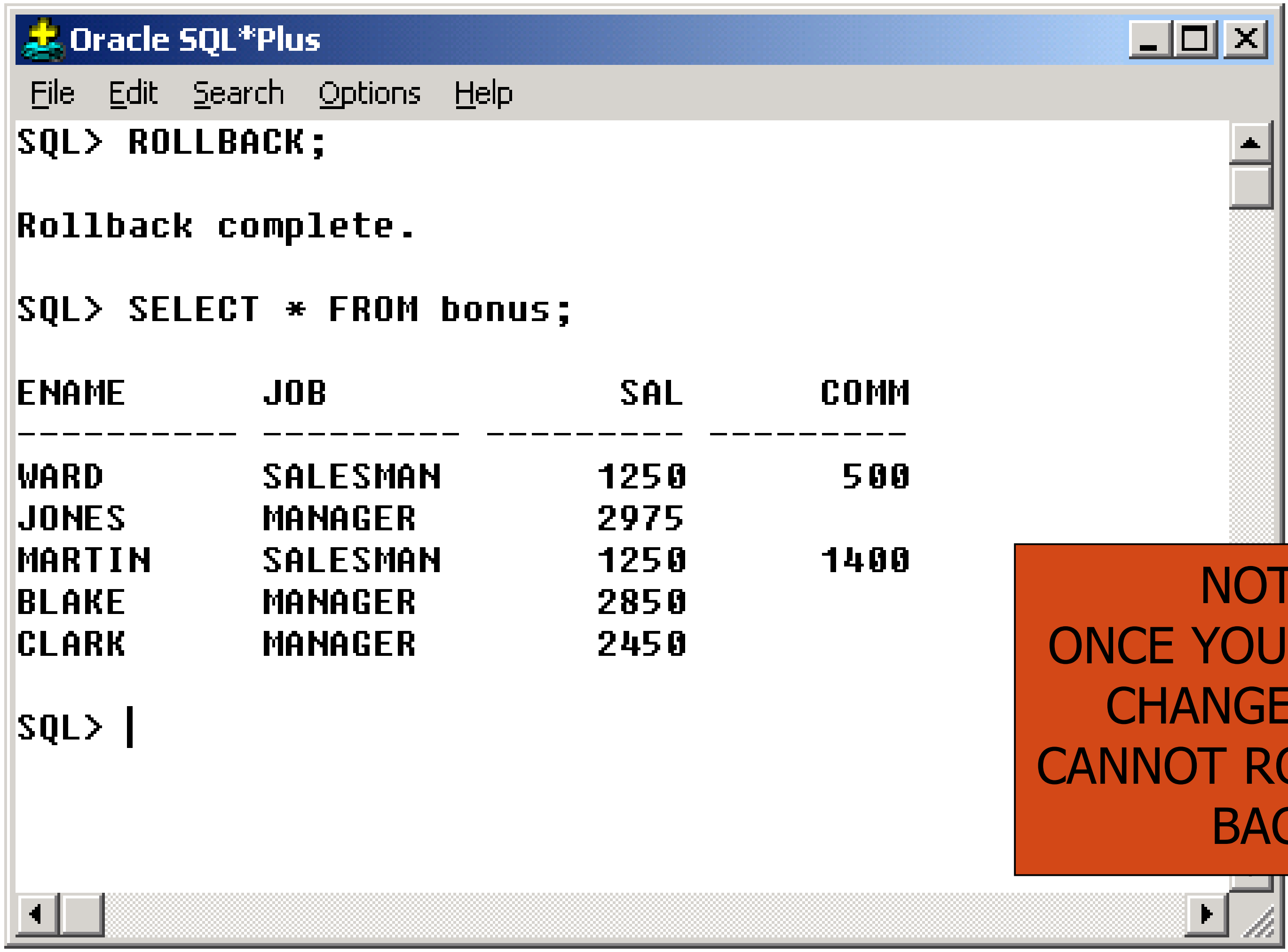
results



```
SQL> INSERT INTO bonus (ename, job, sal, comm)
  2  SELECT ename, job, sal, comm
  3  FROM emp
  4  WHERE job = 'MANAGER';

3 rows created.

SQL> SELECT * FROM bonus;

ENAME      JOB              SAL       COMM
---------- --------- ---------- ----------
WARD       SALESMAN        1250        500
JONES      MANAGER         2975
MARTIN     SALESMAN        1250       1400
BLAKE      MANAGER         2850
CLARK      MANAGER         2450
JONES      MANAGER         2975
BLAKE      MANAGER         2850
CLARK      MANAGER         2450

8 rows selected.

SQL>
```

Making the 'mistake'

Mistake as the managers are already in the table



```
SQL> ROLLBACK;

Rollback complete.

SQL> SELECT * FROM bonus;

ENAME      JOB              SAL       COMM
---------- --------- ---------- ----------
WARD       SALESMAN        1250        500
JONES      MANAGER         2975
MARTIN     SALESMAN        1250       1400
BLAKE      MANAGER         2850
CLARK      MANAGER         2450

SQL> |
```

NOTE:
ONCE YOU COMMIT CHANGES YOU CANNOT ROLL THEM BACK

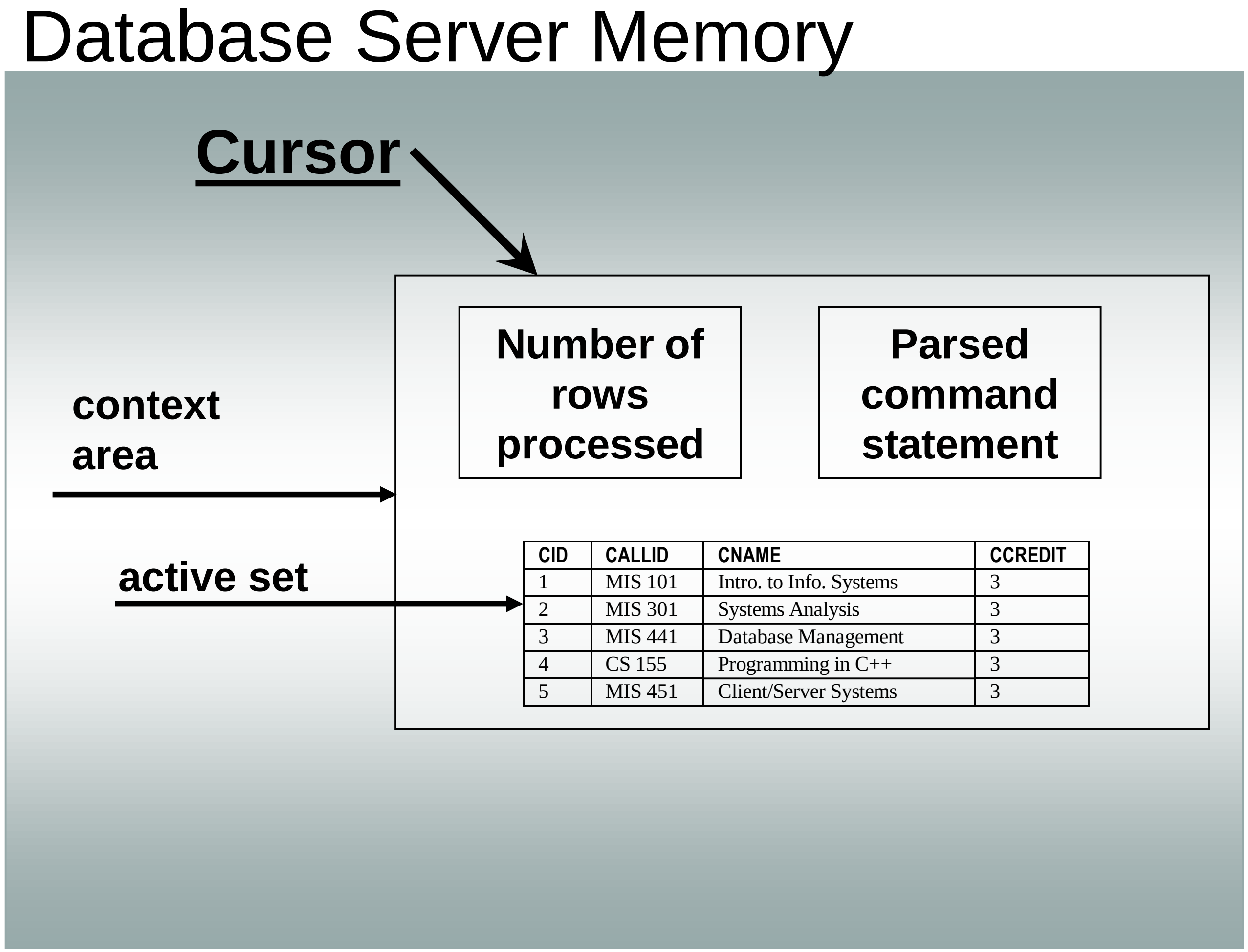
Question

- Write a PL/SQL block of code that updates the salaries of Maria Jacob and Albert by Rs. 2000/- and Rs.2500/- respectively. Then check to see that the total salary does not exceed 75000. If the total salary is greater than 75000, then undo the updates made to salaries of both. (Use savepoint, rollback and commit).

Cursors

- The oracle engine uses a work area for its internal processing in order to execute an SQL statement. This work area is private to SQLs operation
- Contains information about a SQL command in a PL/SQL program
- The data that is stored in the cursor is called the active data set

Cursors



CID	CALLID	CNAME	CCREDIT
1	MIS 101	Intro. to Info. Systems	3
2	MIS 301	Systems Analysis	3
3	MIS 441	Database Management	3
4	CS 155	Programming in C++	3
5	MIS 451	Client/Server Systems	3

Types of Cursors

- Implicit
  - Opened by the oracle engine for its internal processing
- Explicit
  - Userdefined cursor opened on demand by the PL/SQL

Oracle processing of SQL statements

- Reserves a private SQL area in memory
- Populates this area with the data requested in the SQL statement
- Processes the data in this memory as required.
- Frees the memory area when the processing of data is complete

Implicit Cursors

- Created automatically every time you use an INSERT, UPDATE, DELETE, or SELECT command
- Doesn’t need to be declared
- Can be used to access information about the status of last insert, delete, update or single row select statement
- Can be used to assign the output of a SELECT command to one or more PL/SQL variables
- Can only be used if query returns one and only one record

Cursor Attributes

cursorname%ROWCOUNT	Rows returned so far
cursorname%FOUND	One or more rows retrieved
cursorname%NOTFOUND	No rows found
Cursorname%ISOPEN	Is the cursor open

- For implicit cursor attributes, cursor name is SQL
- The value of cursor attributes always refer to the most recently executed SQL statement

Implicit Cursor example

```
Update employee set branchno=&branchno where empno=&empno;
If sql%found then
dbms_output.put_line ('employee successfully transferred')
endif
```

Questions

- The HRD manager decides to raise the salary of employees by 0.15. Write a PL/SQL block to accept an employee number and update the salary of that employee. Display appropriate message based on the existence of the record in the employee table.
- The HRD manager decides to raise the salary of employees working as ‘analyst’ by 0.15. Write a cursor to update the salary of the employees. Display the no. of employee records that has been modified.

The HRD manager decides to raise the salary of employees working as ‘analyst’ by 0.15. Write a cursor to update the salary of the employees. Display appropriate message based on the existence of the record in the employee table.

Employee(empcode,empname,job,salary)

```
declare
    rowsaff number;
begin
    update employee set salary=salary + (salary*0.15)where job='analyst';
    rowsaff:=SQL%rowcount;
    if SQL%rowcount>0 then
            dbms_output.put_line(rowsaff||'employees updated');
    else
            dbms_output.put_line('no employees updated');
    end if;
end;
```

Explicit Cursors

- Must be declared in program DECLARE section
- Can be used to assign the output of a SELECT command to one or more PL/SQL variables
- Can be used if query returns multiple records or no records

Using an Explicit Cursor

- Declare the cursor
- Open the cursor
- Fetch the data from cursor result into PL/SQL program variables
- Process the data as required using loop.
- Exit from loop
- Close the cursor

Declaring an Explicit Cursor

```
DECLARE
  CURSOR cursor_name IS SELECT_statement;
```

- *The 3 commands used to control subsequently are open, fetch and close*

Opening an Explicit Cursor

```
OPEN cursor_name;
```

- *Defines a private SQL area named after the cursor name*
- *Executes a query associated with the cursor*
- *Retrieves table data and populates the named private SQL area in memory(active data set)*
- *Sets the cursor row pointer in the active data set to the first record*

Fetching Explicit Cursor Records

```
FETCH cursor_name INTO
  variable_name(s);
```

- *Moves data held in the active data set to memory variables*
- *Placed inside a Loop…EndLoop construct*
- *Exiting of fetch loop is user controlled.*

Closing an Explicit Cursor

```
CLOSE cursor_name;
```

- *Releases the memory occupied by the cursor and its active data set.*

Processing an Explicit Cursor

- LOOP ..EXIT WHEN approach:

```
OPEN cursor_name;
LOOP
  FETCH cursor_name INTO variable_name(s);
  EXIT WHEN cursor_name%NOTFOUND:
END LOOP;
CLOSE cursor_name;
```

Processing an Explicit Cursor

- Cursor FOR Loop approach:

```
FOR variable_name(s) in cursor_name LOOP
  additional processing statements;
END LOOP;
```

Using Reference Data Types in Explicit Cursor Processing

- Declaring a ROWTYPE reference variable:

```
DECLARE
  reference_variable_name cursor_name%ROWTYPE;
```

- Referencing a ROWTYPE reference variable:

```
reference_variable_name.database_field_name
```

Explicit Cursor Attributes

Attribute	Return Value
%NOTFOUND	TRUE when no rows left to fetch; FALSE when rows left to fetch
%FOUND	TRUE when rows left to fetch; FALSE when no rows left to fetch
%ROWCOUNT	Number of rows a cursor has fetched so far
%ISOPEN	TRUE if cursor is open and FALSE is cursor is closed

Sample Cursor Program

```
DECLARE
  CURSOR students_cursor IS
    SELECT * from students;
  v_student students_cursor%rowtype;
  /* instead we could do v_student students%rowtype */
BEGIN
  DBMS_OUTPUT.PUT_LINE ('******************');
  OPEN students_cursor;
  FETCH students_cursor into v_student;
  WHILE students_cursor%found LOOP
     DBMS_OUTPUT.PUT_LINE (v_student.last);
     DBMS_OUTPUT.PUT_LINE (v_student.major);
     DBMS_OUTPUT.PUT_LINE ('******************');
     FETCH students_cursor into v_student;
  END LOOP;
  CLOSE students_cursor;
END;
/
```

Sample Cursor Program

(same program without composite variable)

```
DECLARE
  CURSOR students_cursor IS
    SELECT last, major from students;
  v_Last students.last%type;
  v_major students.major%type;
BEGIN
  DBMS_OUTPUT.PUT_LINE ('******************');
  OPEN students_cursor;
  FETCH students_cursor into v_last, v_major;
  WHILE students_cursor%found LOOP
     DBMS_OUTPUT.PUT_LINE (v_last);
     DBMS_OUTPUT.PUT_LINE (v_major);
     DBMS_OUTPUT.PUT_LINE ('******************');
     FETCH students_cursor into v_last, v_major;
  END LOOP;
  CLOSE students_cursor;
END;
/
```

Parameterized Cursor

Cursor *cursorname(variable_name datatype)* **is <select** *statement…***>**

Open *cursorname(value/variable/expression)*

Sample Cursors(Explicit)

Write an explicit cursor to display the name,department, salary of the first 5 employees getting the highest salary.

Employee(empcode,empname,job,salary)

```
declare
    Cursor cur_emp is
            select empname,salary from employee order by salary desc;
    ename employee.empname%type;
    sal employee.salary%type;
    Begin
            open cur_emp;
            dbms_output.put_line('name     salary');
            Loop
                    Fetch cur_emp into ename,sal;
                    exit when cur_emp%rowcount=4 or cur_emp%notfound;
                    dbms_output.put_line(ename||'    '||sal);
            end loop;
    close cur_emp;
    end;
```

Question

- The HRD manager decides to raise the salary of employees working as 'analyst' by 0.15. Whenever any such raise is given to the employees, a record for the same is maintained in the emp_raise table. It includes the employee number, the date when the raise was given and actual raise. Write a PL/SQL block to update the salary of the employees and insert a record in the emp_raise table.

Emp_raise(empcode, raisedate,raise_amt)

PROCEDURES, FUNCTIONS & TRIGGERS

PROCEDURES

- A procedure is a module performing one or more actions; it does not need to return any values.
- The syntax for creating a procedure is as follows:

```
CREATE OR REPLACE PROCEDURE name
    [(parameter{IN,OUT,IN
    OUT}DATATYPE)[, parameter, ...])]
AS
    [local declarations]
BEGIN
    executable statements
[EXCEPTION
    exception handlers]
END [name];
```

PROCEDURES

- A procedure may have 0 to many parameters.
- Every procedure has two parts:
    1. The header portion, which comes before AS (sometimes you will see IS—they are interchangeable), keyword (this contains the procedure name and the parameter list),
    2. The body, which is everything after the IS keyword.
- The word REPLACE is optional.
- When the word REPLACE is not used in the header of the procedure, in order to change the code in the procedure, it must be dropped first and then re-created.

Example

```
CREATE OR REPLACE PROCEDURE Discount
AS
   CURSOR c_group_discount
   IS
   SELECT distinct c.course_no,c.course_name FROM
   enrollment e, course c WHERE c.course_no =
   e.course_no
   GROUP BY c.course_no,c.course_name HAVING
   COUNT(*) >=4;
```

Example

```
BEGIN
FOR r_group_discount IN c_group_discount
   LOOP
          UPDATE course SET cost = cost * .95 WHERE
   course_no = r_group_discount.course_no;
          DBMS_OUTPUT.PUT_LINE('A 5% discount has
   been given to'||r_group_discount.course_no||'
   '||r_group_discount.course_name
                    );
   END LOOP;
END;
```

Example

In order to execute a procedure in SQL*Plus use the following syntax:

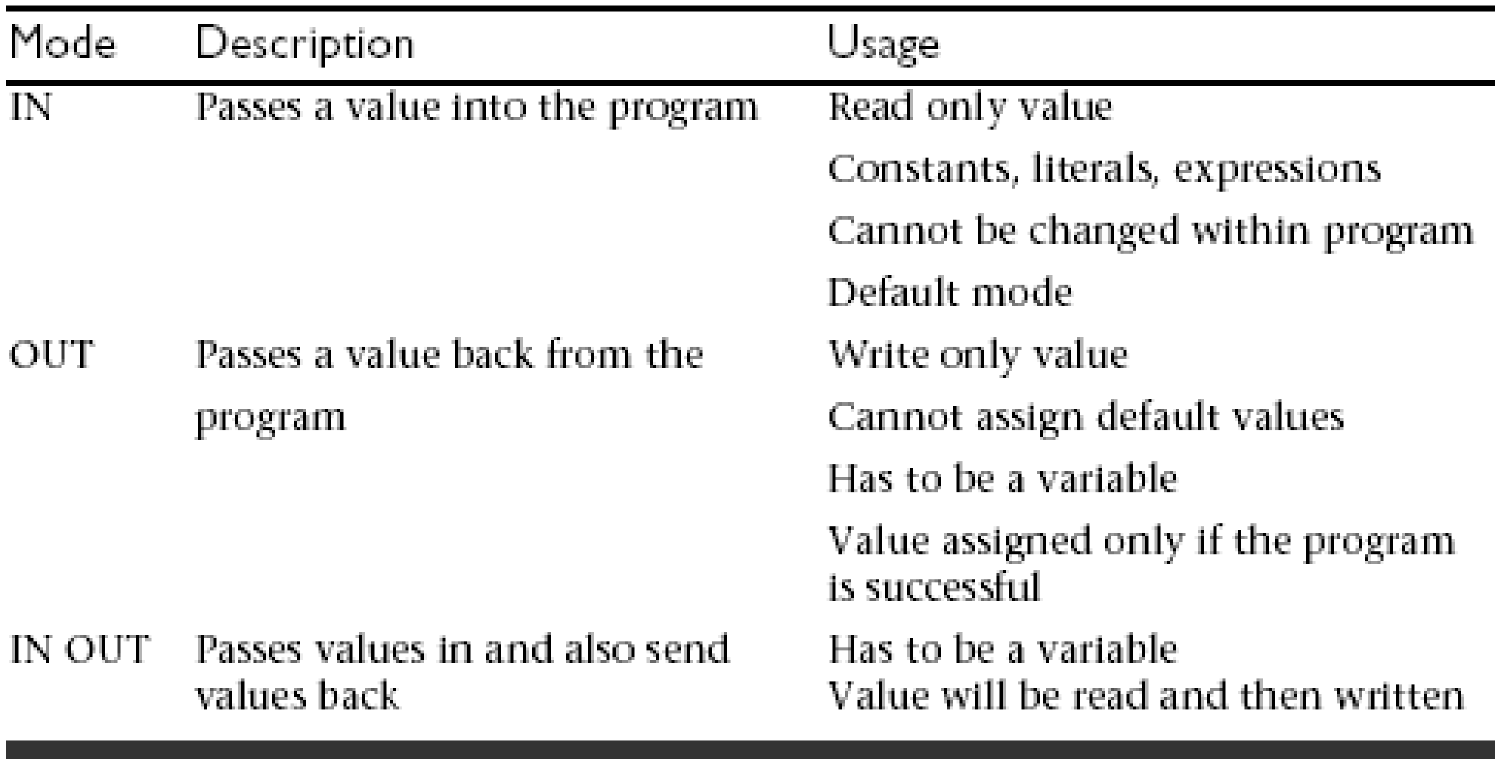
EXECUTE Procedure_name

```
SQL> EXECUTE Discount
```

PARAMETERS

- Parameters are the means to pass values to and from the calling environment to the server.
- These are the values that will be processed or returned via the execution of the procedure.
- There are three types of parameters:
- IN, OUT, and IN OUT.

Types of Parameters



Mode	Description	Usage
IN	Passes a value into the program	Read only value
		Constants, literals, expressions
		Cannot be changed within program
		Default mode
OUT	Passes a value back from the program	Write only value
		Cannot assign default values
		Has to be a variable
		Value assigned only if the program is successful
IN OUT	Passes values in and also send values back	Has to be a variable Value will be read and then written

FORMAL AND ACTUAL PARAMETERS

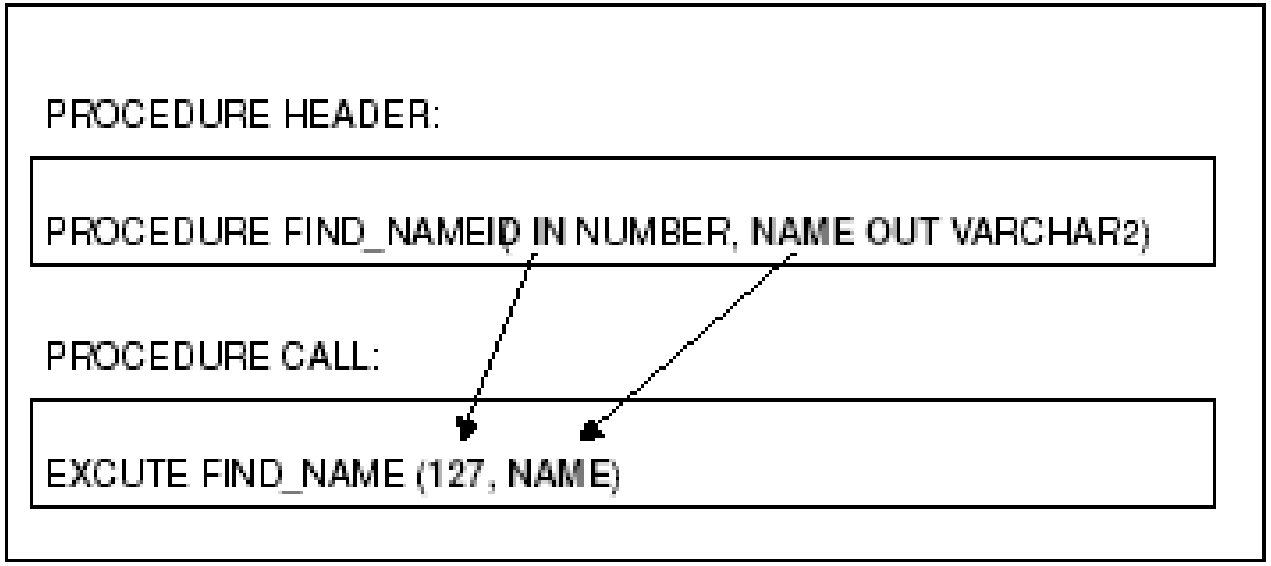
- *Formal parameters* are the names specified within parentheses as part of the header of a module.
- *Actual parameters* are the values—expressions specified within parentheses as a parameter list—when a call is made to the module.
- The formal parameter and the related actual parameter must be of the same or compatible data types.

MATCHING ACTUAL AND FORMAL PARAMETERS

- Two methods can be used to match actual and formal parameters: positional notation and named notation.
- *Positional notation* is simply association by position: The order of the parameters used when executing the procedure matches the order in the procedure's header exactly.
- *Named notation* is explicit association using the symbol =>
  - `Syntax: formal_parameter_name => argument_value`

- In named notation, the order does not matter.
- If you mix notation, list positional notation before named notation.

MATCHING ACTUAL AND FORMAL PARAMETERS



PROCEDURE HEADER:

```
PROCEDURE FIND_NAME(ID IN NUMBER, NAME OUT VARCHAR2)
```

PROCEDURE CALL:

```
EXCUTE FIND_NAME (127, NAME)
```

Questions

- Write a PL/SQL block which makes use of a stored procedure Proj_emp (emp_name varchar2(50)) which finds all the details of the projects involved by the given employee.
- Write a procedure to check whether a string is a palindrome . Call the procedure to list all the palindrome names in the employee table.

FUNCTIONS

- Functions are a type of stored code and are very similar to procedures.
- The significant difference is that a function is a PL/SQL block that *returns* a single value.
- Functions can accept one, many, or no parameters, but a function must have a return clause in the executable section of the function.
- The datatype of the return value must be declared in the header of the function.
- A function is not a stand-alone executable in the way that a procedure is: It must be used in some context. You can think of it as a sentence fragment.
- A function has output that needs to be assigned to a variable, or it can be used in a SELECT statement.

FUNCTIONS

- The syntax for creating a function is as follows:

```
CREATE [OR REPLACE] FUNCTION function_name
        (parameter list)
        RETURN datatype
IS
BEGIN
        <body>
        RETURN (return_value);
END;
```

FUNCTIONS

- The function does not necessarily have to have any parameters, but it must have a RETURN value declared in the header, and it must return values for all the varying possible execution streams.
- The RETURN statement does not have to appear as the last line of the main execution section, and there may be more than one RETURN statement (there should be a RETURN statement for each exception).
- A function may have IN, OUT, or IN OUT parameters. but you rarely see anything except IN parameters.

Example

```
CREATE OR REPLACE FUNCTION show_description
        (i_course_no number)
RETURN varchar2
AS
        v_description varchar2(50);
BEGIN
        SELECT description
                INTO v_description
                FROM course
                WHERE course_no = i_course_no;
        RETURN v_description;
EXCEPTION
        WHEN NO_DATA_FOUND
        THEN
                RETURN('The Course is not in the database');
        WHEN OTHERS
        THEN
                RETURN('Error in running show_description');
END;
```

Making Use Of Functions

- **In a anonymous block**

```
SET SERVEROUTPUT ON
DECLARE
    v_description VARCHAR2(50);
BEGIN
    v_description := show_description(&sv_cnumber);
    DBMS_OUTPUT.PUT_LINE(v_description);
END;
```

- **In a SQL statement**

```
SELECT course_no, show_description(course_no)   FROM
    course;
```

Question

- Write a function to find the reverse of EmpNo in Employee table and display the EmpNo and Reversed(Emp No) of the first 5 employees using an SQL Query.
- Write a function that would check for the existence of an employee in the employee table given an EmpNo. If existing employee, check whether he is the manager of any department and display messages accordingly.

TRIGGERS

- A database trigger is a stored PL/SQL program unit associated with a specific database table. ORACLE executes (fires) a database trigger automatically when a given SQL operation (like INSERT, UPDATE or DELETE) affects the table. Unlike a procedure, or a function, which must be invoked explicitly, database triggers are invoked implicitly.

TRIGGERS

- Database triggers can be used to perform any of the following:
  - Audit data modification
  - Log events transparently
  - Enforce complex business rules
  - Derive column values automatically
  - Implement complex security authorizations
  - Maintain replicate tables

TRIGGERS

- You can associate up to 12 database triggers with a given table. A database trigger has three parts: a **triggering event**, an **optional trigger constraint**, and a **trigger action**.

- When an event occurs, a database trigger is fired, and an predefined PL/SQL block will perform the necessary action.

TRIGGERS

- **SYNTAX:**

CREATE [OR REPLACE] TRIGGER ***trigger_name***
{BEFORE|AFTER} ***triggering_event*** **ON** ***table_name***
[FOR EACH ROW]
[WHEN ***condition*****]**
DECLARE
Declaration statements
BEGIN
Executable statements
EXCEPTION
Exception-handling statements
END;

TRIGGERS

- The trigger_name references the name of the trigger.
- BEFORE or AFTER specify when the trigger is fired (before or after the triggering event).
- The triggering_event references a DML statement issued against the table (e.g., INSERT, DELETE, UPDATE).
- The table_name is the name of the table associated with the trigger.
- The clause, FOR EACH ROW, specifies a trigger is a row trigger and fires once for each modified row.
- A WHEN clause specifies the condition for a trigger to be fired.
- Bear in mind that if you drop a table, all the associated triggers for the table are dropped as well.

Classification Of Triggers

Triggers can be classified based on the following parameters.

- Classification based on the **timing**
  - BEFORE Trigger: It fires before the specified event has occurred.
  - AFTER Trigger: It fires after the specified event has occurred.
  - INSTEAD OF Trigger: used with complex Views
- Classification based on the **level**
  - STATEMENT level Trigger: It fires one time for the specified event statement.
  - ROW level Trigger: It fires for each record that got affected in the specified event. (only for DML)
- Classification based on the **Event**
  - DML Trigger: It fires when the DML event is specified (INSERT/UPDATE/DELETE)
  - DDL Trigger: It fires when the DDL event is specified (CREATE/ALTER)
  - DATABASE/Event Trigger: It fires when the database event is specified (LOGON/LOGOFF/STARTUP/SHUTDOWN)

Classification based on the **timing**

- Triggers may be called BEFORE or AFTER the following events:

  INSERT, UPDATE and DELETE.

- The before/after options can be used to specify when the trigger body should be fired with respect to the triggering statement. If the user indicates a BEFORE option, then Oracle fires the trigger before executing the triggering statement. On the other hand, if an AFTER is used, Oracle fires the trigger after executing the triggering statement.

Classification based on the **level**

- A trigger may be a ROW or STATEMENT type. If the statement FOR EACH ROW is present in the CREATE TRIGGER clause of a trigger, the trigger is a row trigger. A row trigger is fired for each row affected by a triggering statement.
- A statement trigger, however, is fired only once for the triggering statement, regardless of the number of rows affected by the triggering statement

- **Example: statement trigger**

```
CREATE OR REPLACE TRIGGER mytrig1 BEFORE DELETE OR INSERT OR
        UPDATE ON employee
BEGIN
IF (TO_CHAR(SYSDATE, 'day') IN ('sat', 'sun')) OR
        (TO_CHAR(SYSDATE,'hh:mi') NOT BETWEEN '08:30' AND '18:30') THEN
        RAISE_APPLICATION_ERROR(-20500, 'table is secured');
END IF;
END;
/
```

- The above example shows a trigger that limits the DML actions to the employee table to weekdays from 8.30am to 6.30pm. If a user tries to insert/update/delete a row in the EMPLOYEE table, a warning message will be prompted.

Example: ROW Trigger

```
CREATE OR REPLACE TRIGGER mytrig2
AFTER DELETE OR INSERT OR UPDATE ON employee
FOR EACH ROW
BEGIN
IF DELETINGTHEN
        INSERT INTO xemployee (emp_ssn, emp_last_name,emp_first_name, deldate)
        VALUES (:old.emp_ssn, :old.emp_last_name,:old.emp_first_name, sysdate);
ELSIF INSERTINGTHEN
        INSERT INTO nemployee (emp_ssn, emp_last_name,emp_first_name, adddate)
        VALUES (:new.emp_ssn, :new.emp_last_name,:new.emp_first_name, sysdate);
 ELSIF UPDATING('emp_salary')THEN
        INSERT INTO cemployee (emp_ssn, oldsalary, newsalary, up_date)
        VALUES (:old.emp_ssn,:old.emp_salary, :new.emp_salary, sysdate);
ELSE
        INSERT INTO uemployee (emp_ssn, emp_address, up_date)
        VALUES (:old.emp_ssn, :new.emp_address, sysdate);
END IF;
END;
/
```

- **<u>Example: ROW Trigger</u>**

- The previous trigger is used to keep track of all the transactions performed on the employee table. If any employee is deleted, a new row containing the details of this employee is stored in a table called xemployee. Similarly, if a new employee is inserted, a new row is created in another table called nemployee, and so on.
- Note that we can specify the old and new values of an updated row by prefixing the column names with the :OLD and :NEW qualifiers.

```
SQL>  DELETE FROM  employee WHERE emp_last_name
       = 'Joshi';
1 row deleted.
SQL> SELECT * FROM xemployee;


EMP_SSN   EMP_LAST_NAME   EMP_FIRST_NAME DELDATE
------------   ----------------------    ------------------------- ----------------
999333333  Joshi                    Dinesh                  02-MAY-03
```

ENABLING, DISABLING, DROPPING TRIGGERS

SQL>ALTER TRIGGER trigger_name DISABLE;

SQL>ALTER TABLE table_name DISABLE ALL TRIGGERS;

To enable a trigger, which is disabled, we can use the following syntax:

SQL>ALTER TABLE table_name ENABLE trigger_name;

All triggers can be enabled for a specific table by using the following command

SQL> ALTER TABLE table_name ENABLE ALL TRIGGERS;

SQL> DROP TRIGGER trigger_name

Questions

- Consider the table ***Employee***. Write PL/SQL statements to create a trigger when fired checks the operation performed on a table and based on the operation, a variable is assigned the value 'update' or 'delete'. Previous values of the modified record of the table ***Employee*** are stored into the appropriate variables declared and inserted to the audit table A***uditEmployee***.
- Write PL/SQL statements to create a trigger which generates an error messages if the salary is below or beyond the valid range 0-5000 on the employee table. The triggering events are update and insert.

Classification based on the **Event-**DDL Triggers

- Oracle allows you to define triggers that will fire when DDL statements are executed. Simply put, DDL is any SQL statement used to create or modify a database object such as a table or an index. Here are some examples of DDL statements:
  - CREATE TABLE
  - ALTER INDEX
  - DROP TRIGGER
- Each of these statements results in the creation, alteration, or removal of a database object.
- The syntax for creating these triggers is remarkably similar to that of DML triggers, except that the firing events differ, and they are not applied to individual tables.

Syntax

```
CREATE [OR REPLACE] TRIGGER trigger name
 {BEFORE | AFTER } { DDL event} ON {DATABASE |
SCHEMA}
   [WHEN (...)]
DECLARE
Variable declarations
BEGIN
...some code...
END;
```

```
CREATE OR REPLACE TRIGGER hr_audit_tr
AFTER DDL ON SCHEMA
BEGIN
INSERT INTO hr_audit VALUES (
sysdate,user);
END;
/
```

Classification based on the **Event-**Event Database Triggers

Database event triggers fire whenever database-wide events occur.
Need to logon to the database as a user with DBA privileges

There are six database event triggers:

- STARTUP
  Fires when the database is opened.
- SHUTDOWN
  Fires when the database is shut down normally.
- SERVERERROR
  Fires when an Oracle error is raised.
- LOGON
  Fires when an Oracle database session begins.
- LOGOFF
  Fires when an Oracle database session terminates normally.
- DB_ROLE_CHANGE
  Fires when a standby database is changed to be the primary database or vice versa.

Start up Triggers

```
CREATE OR REPLACE TRIGGER startup_audit
AFTER STARTUP ON DATABASE
BEGIN
  INSERT INTO startup_audit VALUES
(
    ora_sysevent,
    SYSDATE,
    TO_CHAR(sysdate, 'hh24:mm:ss')
  );
END;
/
```

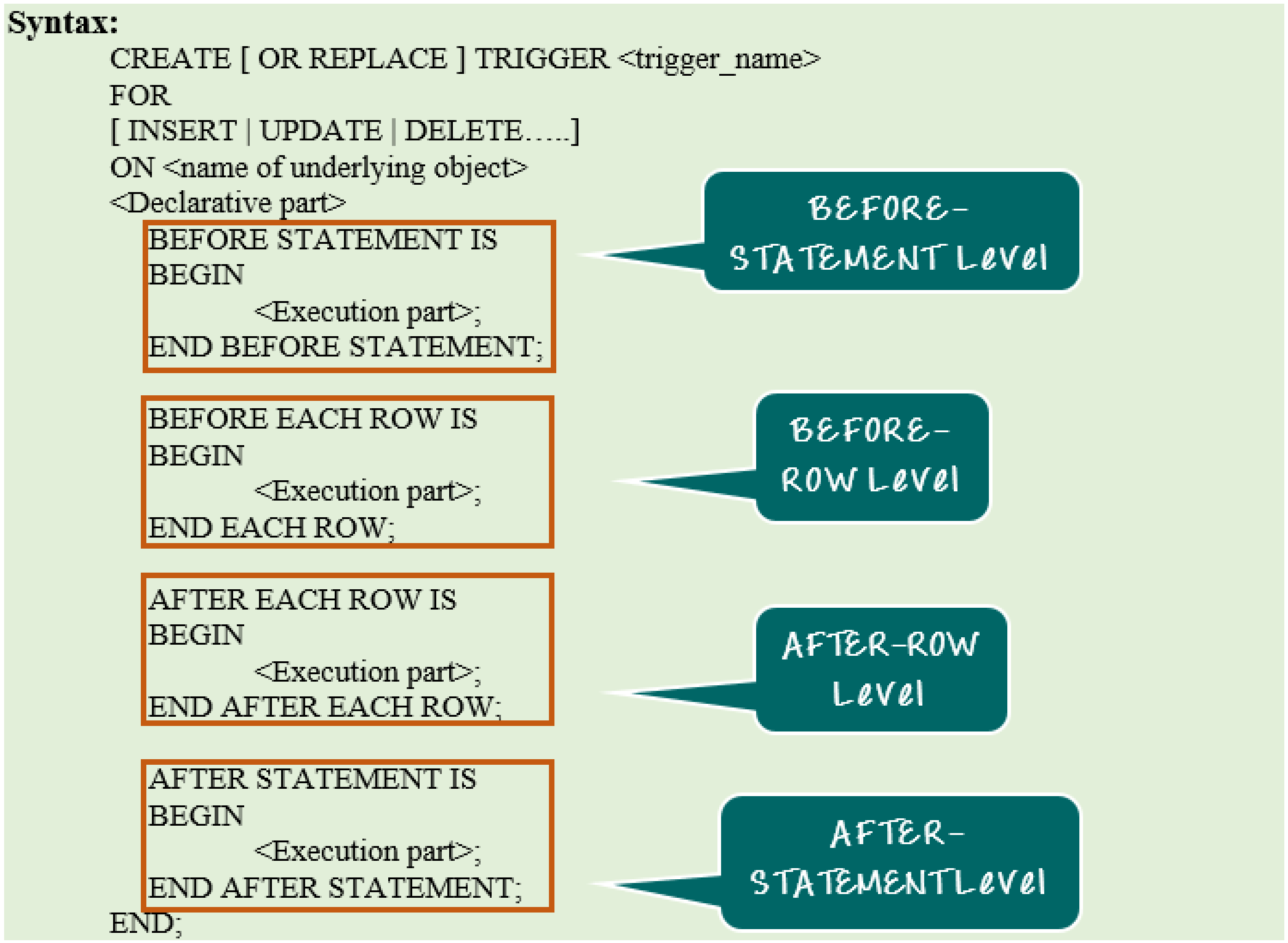
Shut down triggers

```
CREATE OR REPLACE TRIGGER tr_shutdown_audit
BEFORE SHUTDOWN ON DATABASE
BEGIN
  INSERT INTO startup_audit VALUES(
    ora_sysevent,
    SYSDATE,
    TO_CHAR(sysdate, 'hh24:mm:ss')
  );
END;
/
```

Compound Triggers

- A compound trigger is a single trigger on a table that enables you to specify actions for each of four timing points:
  - **Before the firing statement**
  - **Before each row that the firing statement affects**
  - **After each row that the firing statement affects**
  - **After the firing statement**
- With the compound trigger, both the statement-level and row-level action can be put up in a single trigger.
- It allows sharing of common state between all the trigger-points using variable. This is because compound trigger in oracle 11g has a declarative section where one can declare variable to be used within trigger. This common state is established at the start of triggering statement and is destroyed after completion of trigger

- **Compound Triggers Rules**
  - A compound trigger must be a DML trigger.
  - A compound trigger must be defined on either a table or a view.
  - OLD, :NEW, and :PARENT cannot appear in the declarative part, the BEFORE STATEMENT section, or the AFTER STATEMENT section.
  - Only the BEFORE EACH ROW section can change the value of :NEW



Syntax:

```
CREATE [ OR REPLACE ] TRIGGER <trigger_name>
FOR
[ INSERT | UPDATE | DELETE.....]
ON <name of underlying object>
<Declarative part>
   BEFORE STATEMENT IS
   BEGIN
            <Execution part>;
   END BEFORE STATEMENT;

   BEFORE EACH ROW IS
   BEGIN
            <Execution part>;
   END EACH ROW;

   AFTER EACH ROW IS
   BEGIN
            <Execution part>;
   END AFTER EACH ROW;

   AFTER STATEMENT IS
   BEGIN
            <Execution part>;
   END AFTER STATEMENT;
END;
```

```
CREATE TRIGGER emp_trig
FOR INSERT
ON emp
COMPOUND TRIGGER
BEFORE EACH ROW IS
BEGIN
:new.salary:=5000;
END BEFORE EACH ROW;
END emp_trig;
/
```