Contents

[CHAPTER 1 3](#_Toc29806250)

[**1.** **Database Objects** 3](#_Toc29806251)

[**2.** **SQL statements** 3](#_Toc29806252)

[**3.** **Connect to database** 4](#_Toc29806253)

[**4.** **SQL Navigator** 5](#_Toc29806254)

[**5.** **Table in course** 7](#_Toc29806255)

[**6.** **Templates** 8](#_Toc29806256)

[CHAPTER 2 12](#_Toc29806257)

[**1.** **Dual table** 12](#_Toc29806258)

[**2.** **Data types** 12](#_Toc29806259)

[**3.** **Create / Alter table** 13](#_Toc29806260)

[**4.** **Constraints** 15](#_Toc29806261)

[**5.** **Referencing Another User’s Tables** 17](#_Toc29806262)

[**6.** **Practice** 17](#_Toc29806263)

[CHAPTER 3 20](#_Toc29806264)

[**1.** **Select statement** 20](#_Toc29806265)

[**2.** **Where** 20](#_Toc29806266)

[**3.** **Insert statement** 23](#_Toc29806267)

[**4.** **Update statement** 24](#_Toc29806268)

[**5.** **Delete statement** 24](#_Toc29806269)

[**6.** **Joining Table** 25](#_Toc29806270)

[**7.** **Practices** 29](#_Toc29806271)

[CHAPTER 4 34](#_Toc29806272)

[**1.** **Create simple & create complex view** 34](#_Toc29806273)

[**2.** **Drop view** 37](#_Toc29806274)

[**3.** **Using view** 37](#_Toc29806275)

[**4.** **Practices** 37](#_Toc29806276)

[CHAPTER 5 39](#_Toc29806277)

[**1.** **Sequence** 39](#_Toc29806278)

[**2.** **Index** 41](#_Toc29806279)

[**3.** **Synonyms** 41](#_Toc29806280)

[**4.** **Practices:** 42](#_Toc29806281)

[CHAPTER 6 43](#_Toc29806282)

[**1.** **PL/SQL** 43](#_Toc29806283)

[**2.** **Variables** 43](#_Toc29806284)

[**3.** **Control Structures** 46](#_Toc29806285)

[**a.** **Conditional Control** 46](#_Toc29806286)

[**c.** **Sequential Control** 50](#_Toc29806287)

[**i.** **Goto** 50](#_Toc29806288)

[**ii.** **Null statements** 51](#_Toc29806289)

[**4.** **Handing Exceptions** 52](#_Toc29806290)

[**5.** **Practices** 54](#_Toc29806291)

[CHAPTER 7 57](#_Toc29806292)

[**1.** **Single-row function** 57](#_Toc29806293)

[**2.** **Multiple-row function** 63](#_Toc29806294)

[**3.** **User-defined function** 64](#_Toc29806295)

[**4.** **Practices** 66](#_Toc29806296)

[CHAPTER 8 68](#_Toc29806297)

[**1.** **Introduction Stored Procedure** 68](#_Toc29806298)

[**2.** **Create Stored Procedures** 68](#_Toc29806299)

[**3.** **Invoking** 69](#_Toc29806300)

[**4.** **Practices** 70](#_Toc29806301)

[CHAPTER 9 71](#_Toc29806302)

[**1.** **GRANT statement** 71](#_Toc29806303)

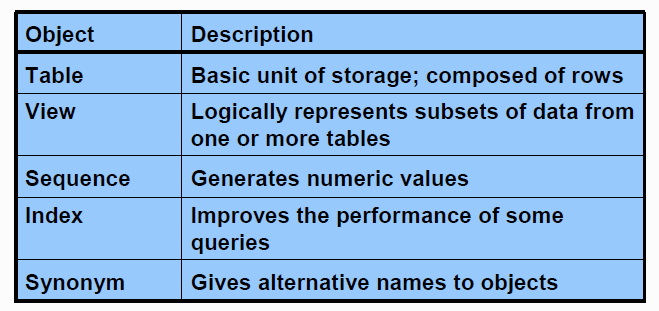
[**2.** **Revoke** 71](#_Toc29806304)

[**3.** **Job** 72](#_Toc29806305)

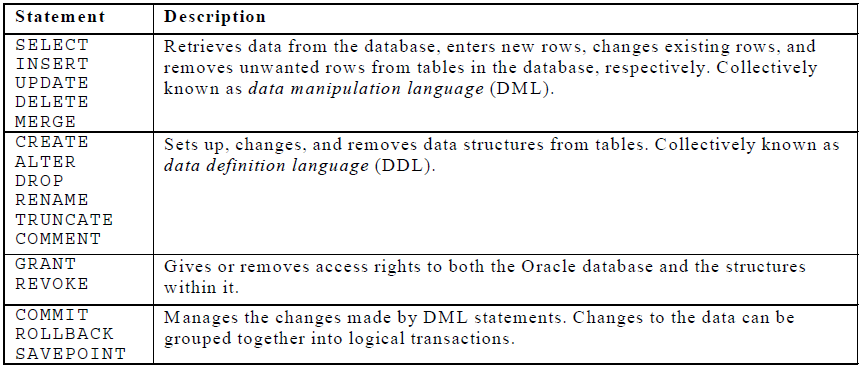
[**4.** **Practices** 75](#_Toc29806306)

# CHAPTER 1

1. **Database Objects**

****

1. **SQL statements**



**COMMIT**       :    Make changes done in  transaction permanent.

*Example*

*insert into emp (empno,ename,sal) values (101,’Abid’,2300);*

*commit;***ROLLBACK**:    Rollbacks the state of database to the last commit point.

To rollback the changes done in a transaction give rollback statement. Rollback restore the state of the database to the last commit point.

*Example :*

*delete from emp;*

*rollback;          /\* undo the changes \*/*

**SAVEPOINT**:    Use to specify a point in transaction to which later you can rollback.

Specify a point in a transaction to which later you can roll back.

*Example*

*insert into emp (empno,ename,sal) values (109,’Sami’,3000);  
savepoint a;  
insert into dept values (10,’Sales’,’Hyd’);  
savepoint b;  
insert into salgrade values (‘III’,9000,12000);*

Now if you give

rollback to a;

Then  row from salgrade table and dept will be roll backed. At this point you can commit the row inserted into emp table or rollback the transaction.

If you give

rollback to b;

Then row inserted into salgrade table will be roll backed. At this point you can commit the row inserted into dept table and emp table or rollback to savepoint a or completely roll backed the transaction.

If you give

rollback;

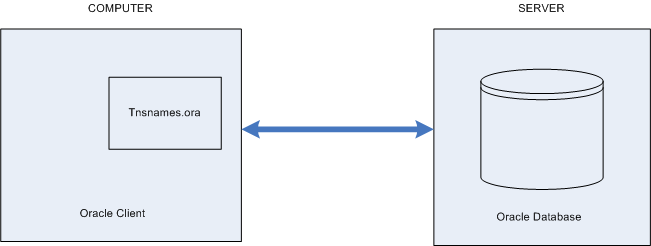
Then the whole transactions is roll backed.

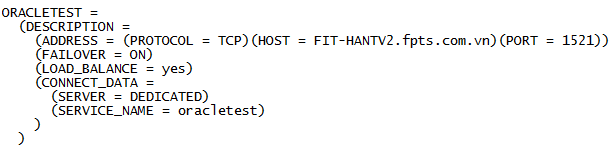
If you give

commit;

Then the whole transaction is committed and all savepoints are removed.

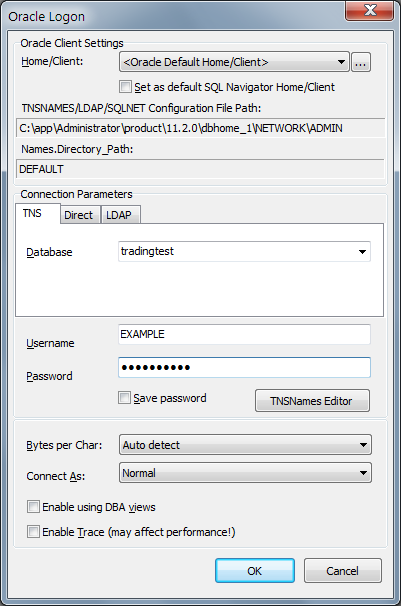
1. **Connect to database**



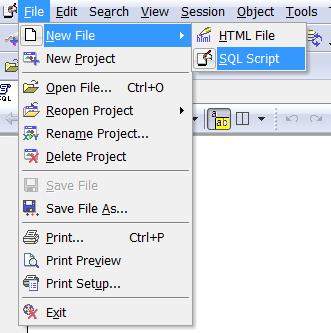


1. **SQL Navigator**

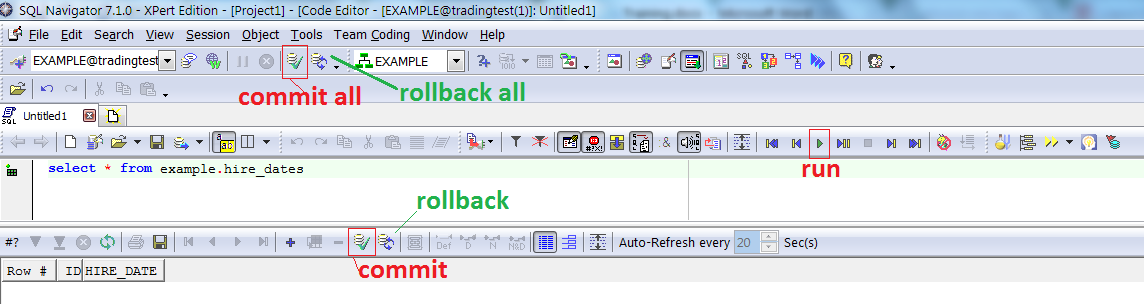
* Log in



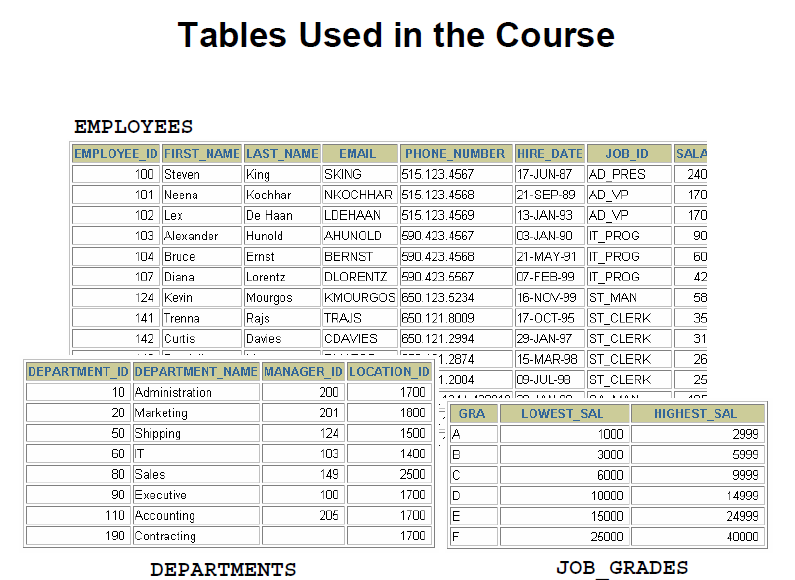
* Open file



* Tool Bar



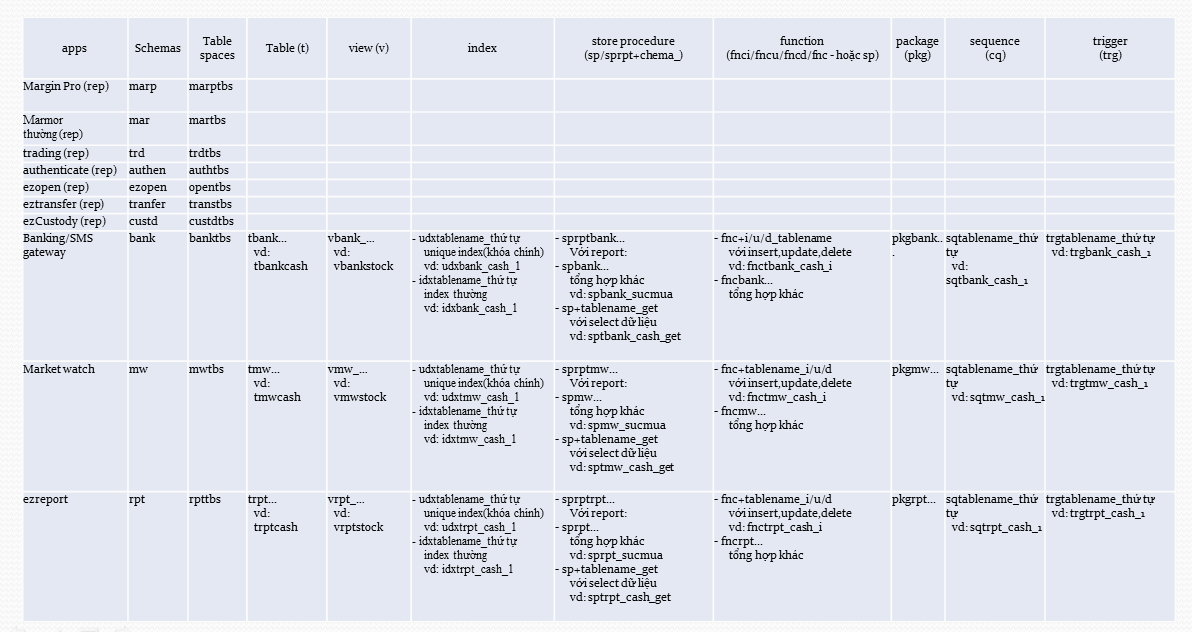
1. **Table in course**



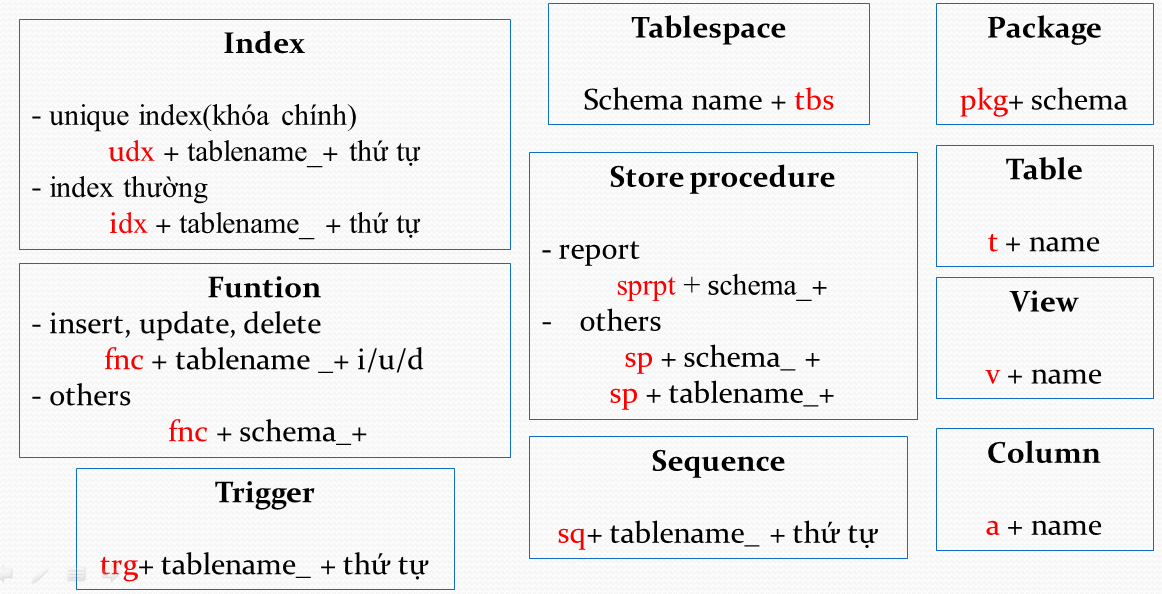
The following main tables are used in this course:

* EMPLOYEES table: Gives details of all the employees
* DEPARTMENTS table: Gives details of all the departments
* JOB\_GRADES table: Gives details of salaries for various grades

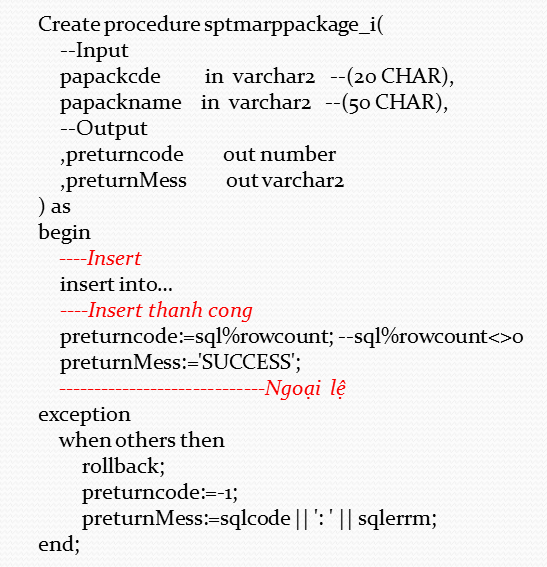
1. **Templates**

****

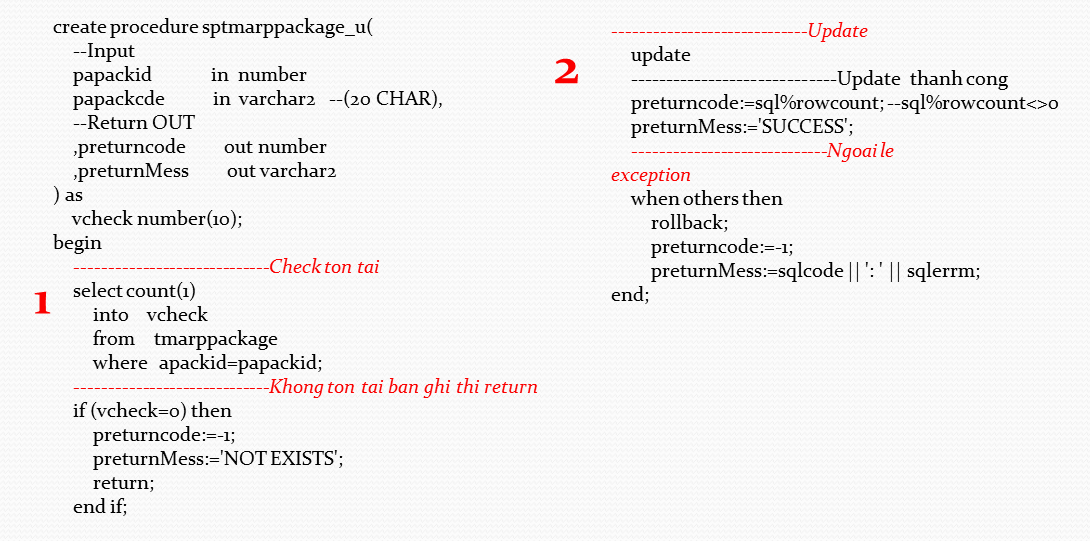
* **Name**

****

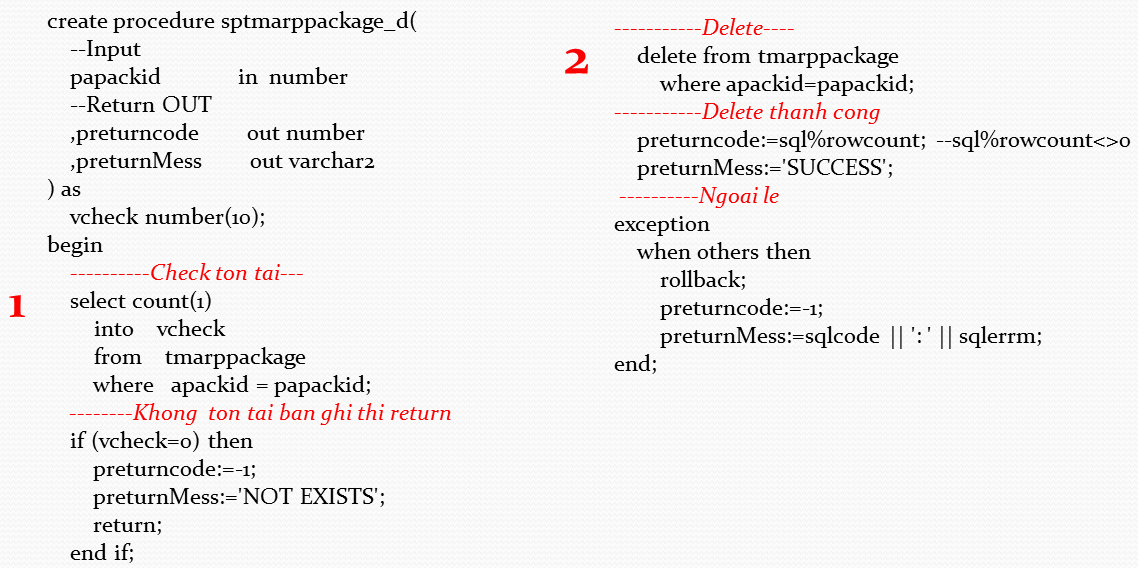
* **Statements**
  + Insert



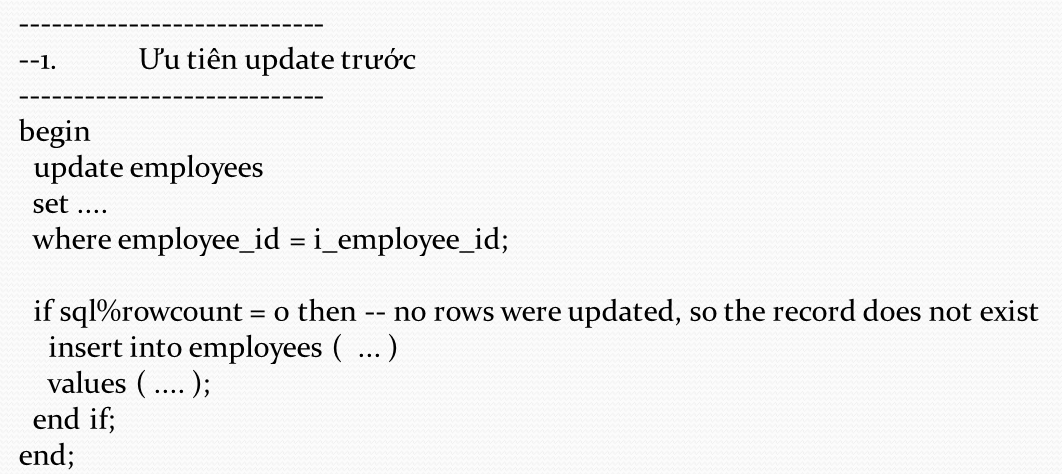
* + Update

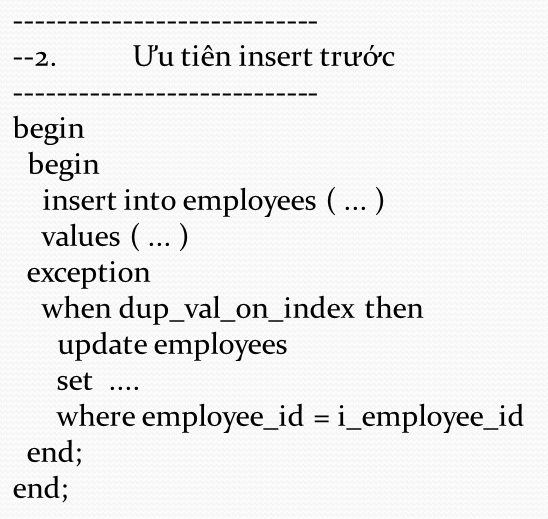


* + Delete



* + Insert & Update





Xem link chi tiết:

* Convention: \\file-server\FPTS-Data\FIT\Projects\App.Training\Dev\HaNTV\Template\namingConventionOraclePublic.xlsx
* Templates: [\\file-server\FPTS-Data\FIT\Projects\App.Training\Dev\HaNTV\Template](file:///\\file-server\FPTS-Data\FIT\Projects\App.Training\Dev\HaNTV\Template)

# CHAPTER 2

1. **Dual table**

DUAL is a dummy table that you can use to view results from functions and calculations.

1. **Data types**

|  |  |
| --- | --- |
| VARCHAR2(*size* [BYTE | CHAR]) | Variable-length character string having maximum length *size* bytes or characters. Maximum *size* is 4000 bytes or characters, and minimum is 1 byte or 1 character. You must specify *size* for VARCHAR2.  BYTE indicates that the column will have byte length semantics. CHAR indicates that the column will have character semantics. |
| NVARCHAR2(*size*) | Variable-length Unicode character string having maximum length *size* characters. The number of bytes can be up to two times *size* for AL16UTF16 encoding and three times *size* for UTF8 encoding. Maximum *size* is determined by the national character set definition, with an upper limit of 4000 bytes. You must specify *size* for NVARCHAR2. |
| CHAR [(*size* [BYTE | CHAR])] | Fixed-length character data of length *size* bytes or characters. Maximum *size* is 2000 bytes or characters. Default and minimum *size* is 1 byte.  BYTE and CHAR have the same semantics as for VARCHAR2. |
| NCHAR[(*size*)] | Fixed-length character data of length *size* characters. The number of bytes can be up to two times *size* for AL16UTF16 encoding and three times *size* for UTF8 encoding. Maximum *size* is determined by the national character set definition, with an upper limit of 2000 bytes. Default and minimum *size* is 1 character. |
| NUMBER [ (*p* [, *s*]) ] | Number having precision *p* and scale *s*. The precision *p* can range from 1 to 38. The scale *s* can range from -84 to 127. Both precision and scale are in decimal digits. A NUMBER value requires from 1 to 22 bytes. |
| DATE | Valid date range from January 1, 4712 BC, to December 31, 9999 AD. The default format is determined explicitly by the NLS\_DATE\_FORMAT parameter or implicitly by the NLS\_TERRITORY parameter. The size is fixed at 7 bytes. This datatype contains the datetime fields YEAR, MONTH, DAY, HOUR, MINUTE, and SECOND. It does not have fractional seconds or a time zone. |
| TIMESTAMP [(*fractional\_seconds\_precision*)] | Year, month, and day values of date, as well as hour, minute, and second values of time, where *fractional\_seconds\_precision* is the number of digits in the fractional part of the SECOND datetime field. Accepted values of *fractional\_seconds\_precision* are 0 to 9. The default is 6. The default format is determined explicitly by the NLS\_DATE\_FORMAT parameter or implicitly by the NLS\_TERRITORY parameter. The sizes varies from 7 to 11 bytes, depending on the precision. This datatype contains the datetime fields YEAR, MONTH, DAY, HOUR, MINUTE, and SECOND. It contains fractional seconds but does not have a time zone. |
| CLOB | A character large object containing single-byte or multibyte characters. Both fixed-width and variable-width character sets are supported, both using the database character set. Maximum size is (4 gigabytes - 1) \* (database block size). |
| BLOB | A binary large object. Maximum size is (4 gigabytes - 1) \* (database block size). |

**Guidelines**

A LONG column is not copied when a table is created using a subquery.

A LONG column cannot be included in a GROUP BY or an ORDER BY clause.

Only one LONG column can be used per table.

No constraints can be defined on a LONG column.

You might want to use a CLOB column rather than a LONG column.

1. **Create / Alter table**

* Create:

Syntax:

**CREATE TABLE [*schema*.]*table***

**(*column datatype* [DEFAULT *expr*][, ...]);**

*schema* is the same as the owner’s name

*table* is the name of the table

DEFAULT *expr* specifies a default value if a value is omitted in the INSERT

statement

*column* is the name of the column

*datatype* is the column’s data type and length

Example:

**CREATE** **TABLE** job\_grades

(grade **CHAR**(1 **BYTE**) ,

lowest\_sal **NUMBER**(8,2) **NOT** **NULL**,

highest\_sal **NUMBER**(8,2) **NOT** **NULL**

)

Create table with sub-query

**CREATE TABLE [*schema*.]*table***

**AS**

**SELECT ……;**

Example:

**create** **table** example.hire\_dates\_bk

**as**

**select** \* **from** example.hire\_dates

* Alter
  + Add a new column
  + Modify an existing column
  + Define a default value for the new column
  + Drop a column

Syntax:

alter table  
   table\_name  
add  
   (  
   column1\_name column1\_datatype column1\_constraint,    
   column2\_name column2\_datatype column2\_constraint,  
   column3\_name column3\_datatype column3\_constraint  
   );

Example:

alter table   
   author   
add   
   (author\_last\_published date,  
    author\_item\_published varchar2(40));

* Drop table

The DROP TABLE statement removes the definition of an Oracle table. When you drop a table, the database loses all the data in the table and all the indexes associated with it.

**Syntax**

DROP TABLE *table*

In the syntax, *table* is the name of the table.

* Confirm

Syntax:

Describe table

Example

describe job\_grades

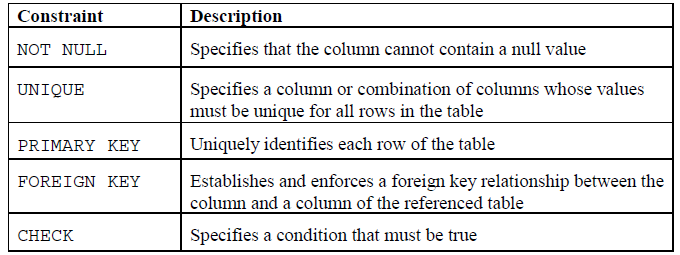
1. **Constraints**

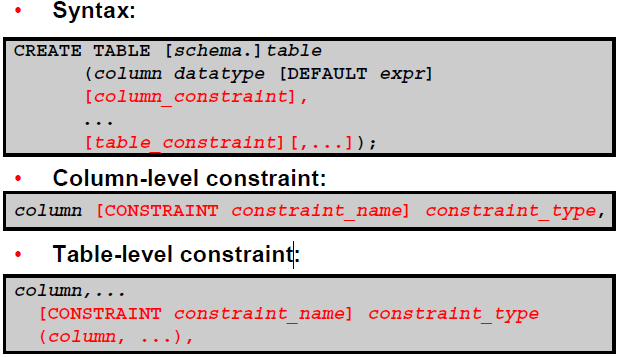
Constraints enforce rules at the table level.

Constraints prevent the deletion of a table if there are dependencies.

The following constraint types are valid:

* NOT NULL
* UNIQUE
* PRIMARY KEY
* FOREIGN KEY
* CHECK



****

In the syntax:

schema is the same as the owner’s name

table is the name of the table

DEFAULT expr specifies a default value to use if a value is omitted in the

INSERT statement

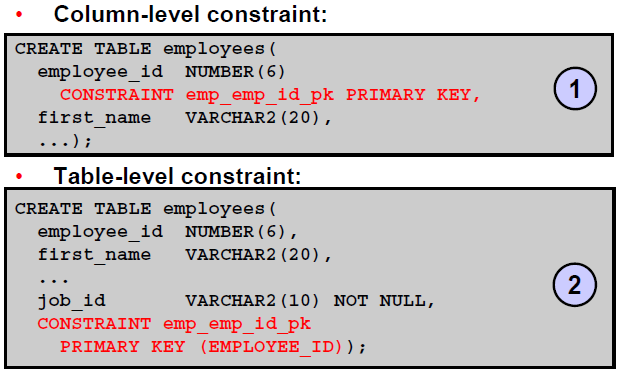
column is the name of the column

datatype is the column’s data type and length

column\_constraint is an integrity constraint as part of the column definition

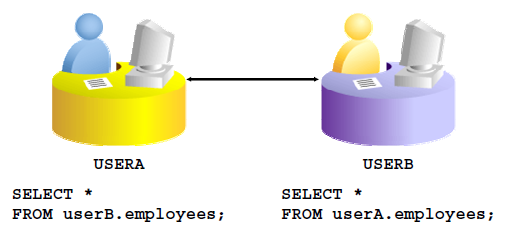
table\_constraint is an integrity constraint as part of the table definition

Example

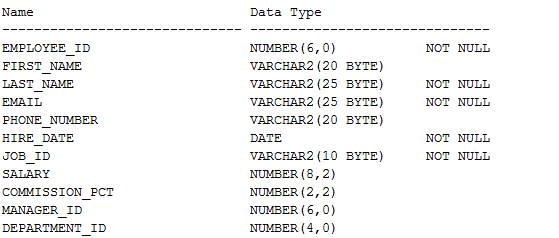


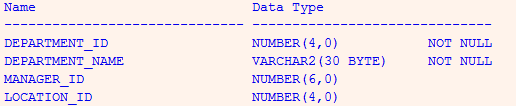
1. **Referencing Another User’s Tables**

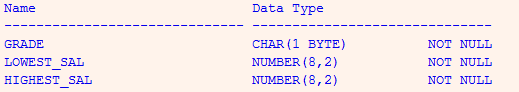
* Tables belonging to other users are not in the user’s schema.
* You should use the owner’s name as a prefix to those tables.



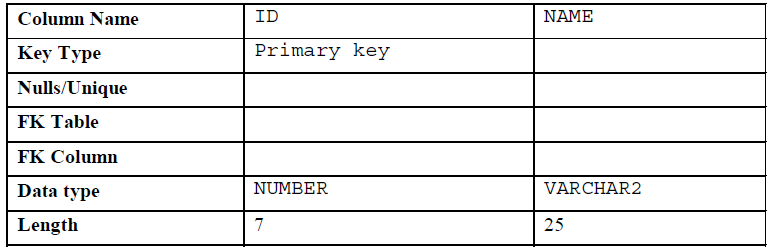
1. **Practice**
   1. Create the Employee, Department, Job\_Grades table based on the following table and confirm table is created



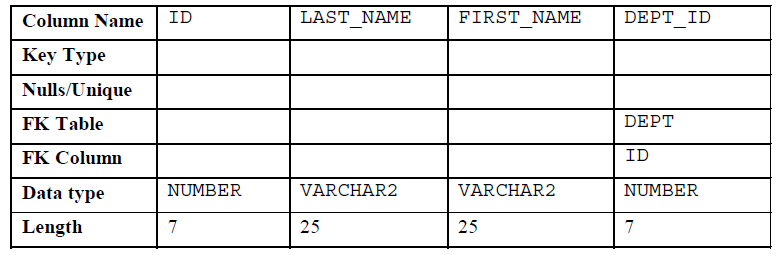




* 1. Create table Dept table based on the following table and confirm table is created



* 1. Create table Emp table based on the following table and confirm table is created.



* 1. Alter table Emp table add “Email” column. Confirm table is altered
  2. Create table Emp2 table based on the structure of the Employees table. Include only the EMPLOYEE\_ID, FIRST\_NAME, LAST\_NAME, SALARY, and DEPARTMENT\_ID columns. Name the columns in your new table ID, FIRST\_NAME, LAST\_NAME, SALARY , and DEPT\_ID
  3. Drop the Emp table.
  4. Write statement to insert data from excel file: Insert\_Data.xlsx

[\\file-server\FPTS-Data\FIT\Projects\App.Training\Dev\HaNTV\Insert\_Data.xlsx](file:///\\file-server\FPTS-Data\FIT\Projects\App.Training\Dev\HaNTV\Insert_Data.xlsx)

# CHAPTER 3

1. **Select statement**

* Objectives: to extract data from the database
* Syntax:

**SELECT \*|{[DISTINCT] *column*|*expression* [*alias*],...}**

**FROM *table;***

In the syntax:

SELECT is a list of one or more columns

\* selects all columns

DISTINCT suppresses duplicates

*column|expression* selects the named column or the expression

*alias* gives selected columns different headings

FROM *table*

Example:

**select** \* **from** employees

1. **Where**

* Objectives:
  + Restrict the rows that are returned by using the WHERE clause
  + The WHERE clause follows the FROM clause
* Syntax:

**SELECT \*|{[DISTINCT] *column|expression* [*alias*],...}**

**FROM *table***

**[WHERE *condition(s)*];**

In the syntax:

WHERE restricts the query to rows that meet a condition

*condition* is composed of column names, expressions, constants, and a comparison operator

Example:

**SELECT** employee\_id, last\_name, job\_id, department\_id

**FROM** employees

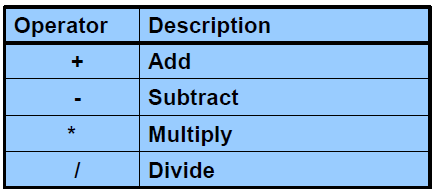
**WHERE** department\_id = 90 ;

**SELECT** \*

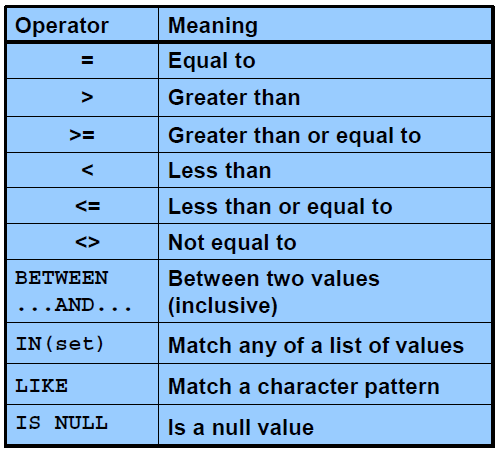
**FROM** employees

**WHERE** hire\_date>='17-jun-1989';

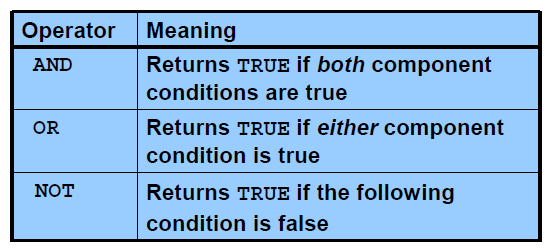
* Arithmetic operators



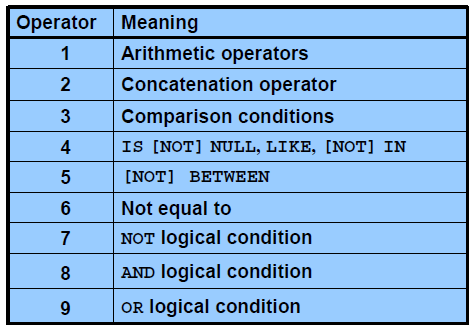
* Comparison Conditions



* Logical Conditions



* Rules of Precedence



Example:

**SELECT** last\_name, job\_id, salary

**FROM** employees

**WHERE** job\_id = 'SA\_REP'

**OR** job\_id = 'AD\_PRES'

**AND** salary > 15000;

* The first condition is that the job ID is AD\_PRES *and* the salary is greater than $15,000.
* The second condition is that the job ID is SA\_REP.
* Sortingc

Sort retrieved rows with the ORDER BY clause:

* + ASC: ascending order, default
  + DESC: descending order

The ORDER BY clause comes last in the SELECT statement

**Syntax**

SELECT *expr*

FROM *table*

[WHERE *condition(s)*]

[ORDER BY {*column*, *expr, numeric\_position*} [ASC|DESC]];

In the syntax:

ORDER BY specifies the order in which the retrieved rows are displayed

ASC orders the rows in ascending order (this is the default order)

DESC orders the rows in descending order

**The default sort order is ascending:**

* Numeric values are displayed with the lowest values first (for example, 1 to 999).
* Date values are displayed with the earliest value first (for example, 01-JAN-92 before
* 01-JAN-95).
* Character values are displayed in alphabetical order (for example, A first and Z last).
* Null values are displayed last for ascending sequences and first for descending sequences.
* You can sort by a column that is not in the SELECT list.

1. **Insert statement**

* Add new rows to a table by using the INSERT statement
* With this syntax, only one row is inserted at a time.
* Syntax:

**INSERT INTO *table* [(*column* [*, column...*])]**

**VALUES *(value* [*, value...*]);**

In the syntax:

*table* is the name of the table

*column* is the name of the column in the table to populate

*value* is the corresponding value for the column

* Example:

**INSERT** **INTO** job\_grades **VALUES** ('A', 1000, 2999);

**Insert** **into** job\_grades

**Select** \* **from** job\_grades\_bk

1. **Update statement**

* Modify existing rows with the UPDATE statement
* Update more than one row at a time (if required).
* Syntax:

**UPDATE *table***

**SET *column* = *value* [, *column* = *value, ...*]**

**[WHERE *condition*];**

In the syntax:

*table* is the name of the table

*column* is the name of the column in the table to populate

*value* is the corresponding value or subquery for the column

*condition* identifies the rows to be updated and is composed of column names, expressions, constants, subqueries, and comparison operators

* Example:

**UPDATE** employees

SET department\_id = 70

**WHERE** employee\_id = 113;

**UPDATE** copy\_emp

SET department\_id = 110;

**UPDATE** employees –- update with subquery

SET

job\_id = (**SELECT** job\_id **FROM** employees **WHERE** employee\_id = 205),

salary = (**SELECT** salary **FROM** employees **WHERE** employee\_id = 205)

**WHERE** employee\_id = 114;

1. **Delete statement**

You can remove existing rows from a table by using the DELETE statement

* Syntax:

**DELETE [FROM] *table***

**[WHERE *condition*];**

In the syntax:

*table* is the table name

*condition* identifies the rows to be deleted and is composed of column names,

expressions, constants, subqueries, and comparison operators

* Example:

**DELETE** **FROM** departments

**WHERE** department\_name = 'Finance';

**DELETE** **FROM** copy\_emp;

1. **Joining Table**

* Equijoins: Equijoins are also called *simple joins* or *inner joins.* Equal Join is a kind of join where condition used is equal(=) sign

Syntax:

**SELECT *table1.column, table2.column***

**FROM *table1, table2***

**WHERE *table1.column1* = *table2.column2;***

In the syntax:

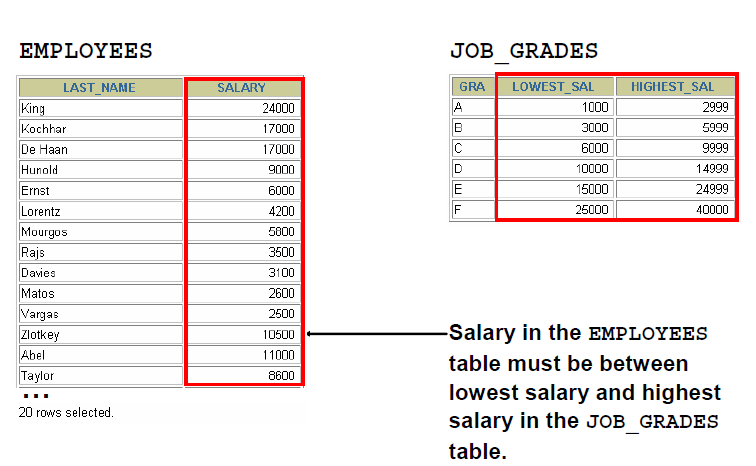
*table1.column* denotes the table and column from which data is retrieved

*table1.column1* = is the condition that joins (or relates) the tables together

*table2.column2*

* Non-Equijoins

An nonequi join is an [inner join](http://www.orafaq.com/wiki/Inner_join) statement that uses an unequal operation (i.e: <>, >, <, !=, BETWEEN, etc.) to match [rows](http://www.orafaq.com/wiki/Row) from different [tables](http://www.orafaq.com/wiki/Table)



Example:

**SELECT** e.last\_name, e.salary, j.grade\_level

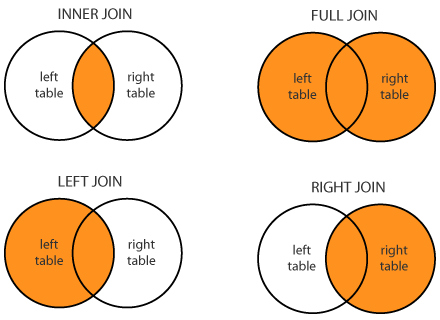
**FROM** employees e, job\_grades j

**WHERE** e.salary

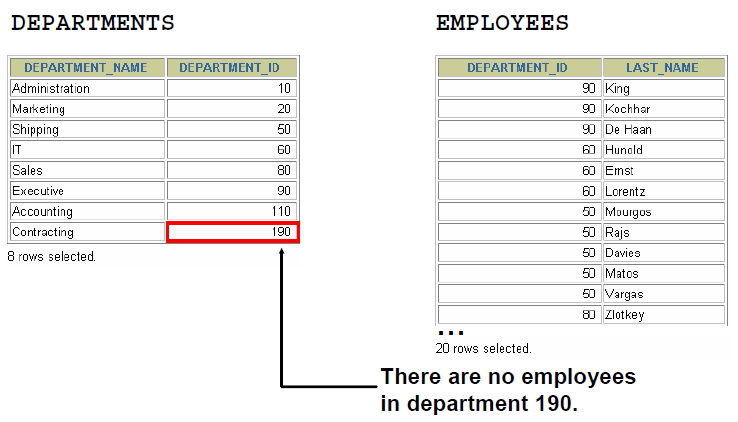
**BETWEEN** j.lowest\_sal **AND** j.highest\_sal;

* Outer Joins

You use an outer join to see rows that do not meet the join condition.



The outer join operator is the plus sign (+).



Syntax:

**SELECT *table1.column, table2.column***

**FROM *table1, table2***

**WHERE *table1.column(+)* = *table2.column;***

**SELECT *table1.column, table2.column***

**FROM *table1, table2***

**WHERE *table1.column* = *table2.column(+);***

In the syntax:

*table1.column =* is the condition that joins (or relates) the tables together

*table2.column* (+) is the outer join symbol, which can be placed on either side of the WHERE clause condition, but not on both sides. (Place the outer join symbol following the name of the column in the table without the matching rows.)

Example:

**select** e.employee\_id, e.last\_name, e.department\_id, d.department\_name

**from** employees e **left** **outer** **join** departments d

**on** e.department\_id = d.department\_id

**where** last\_name = 'Grant';

**select** e.employee\_id, e.last\_name, e.department\_id, d.department\_name

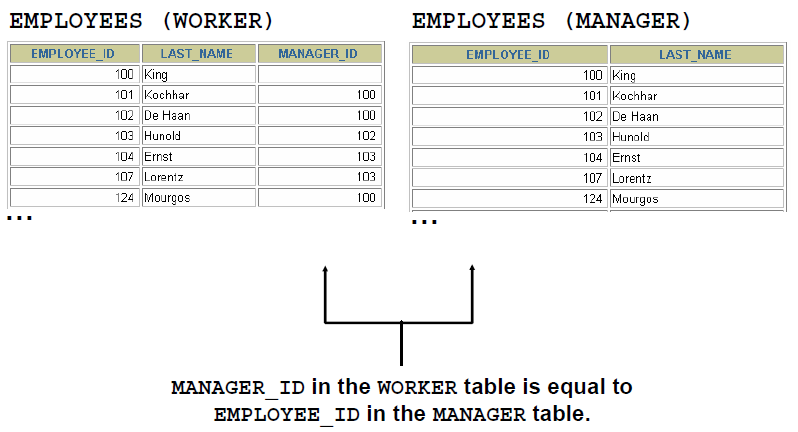
**from** employees e, departments d

**where** e.department\_id = d.department\_id(+) **and** last\_name = 'Grant';

* Self-Joins: Sometimes you need to join a table to itself.

Example:

To find the name of each employee’s manager, you need to join the EMPLOYEES table to itself; this type of join is called a *self-join.*



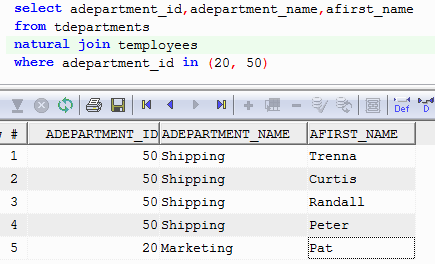
**SELECT** worker.last\_name || ' works for ' || manager.last\_name

**FROM** employees worker, employees manager

**WHERE** worker.manager\_id = manager.employee\_id ;

* NATURAL JOIN clause is based on all columns in the two tables that have the same name

Note: The join can happen on only those columns that have the same names and data types in both tables. If the columns have the same name but different data types, then the NATURAL JOIN syntax causes an error.



1. **Practices**
   1. There are four coding errors in the following statement. Can you identify them?

SELECT employee\_id, last\_name

sal x 12 ANNUAL SALARY

FROM employees;

* There is no comma after the "last\_name" column name in the SELECT clause.
* The multiplication operator is missing an operand. It should be "sal \* 12" instead of "sal x 12".
* There is no alias specified for the calculated column "sal x 12".
* The SQL statement is missing a semicolon at the end of the statement.
  1. The following select statement executes successfully: True/False

**SELECT** last\_name, job\_id, salary **AS** Sal

**FROM** employees; True

* 1. Result of following statement

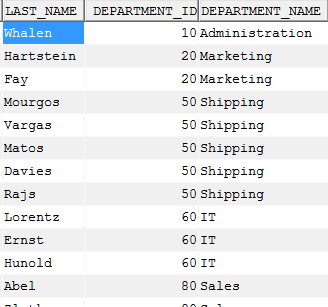
**select** **null** \* 2 **from** dual (Null data)

**select** **null** || 'name' **from** dual (Null data)

* 1. The HR department needs a report of all employees. Write a query to display the last name, department number, and department name for all employees.

- SELECT e.last\_name,e.department\_id AS department\_number,d.department\_name

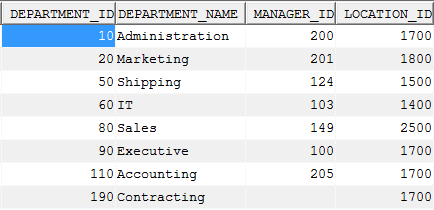
FROM employee e INNER JOIN department d ON e.department\_id = d.department\_id ORDER BY e.department\_id



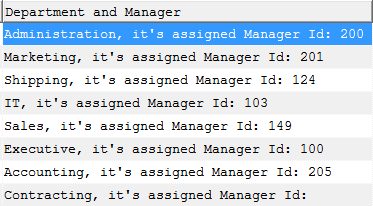
* 1. The HR department needs a report of all department with the following format: **Department\_Name + “, it's assigned Manager Id:” + Manager\_ID**

Example: “Administration, it's assigned Manager Id: 200”

- SELECT department\_name || ', it''s assigned Manager Id: ' || manager\_id AS Department\_And\_Manager

FROM department;

=> Result:



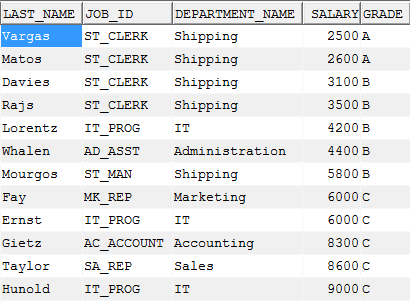
* 1. The HR department needs a report on job grades and salaries. To familiarize yourself with the JOB\_GRADES table, first show the structure of the JOB\_GRADES table. Then create a query that displays the name, job, department name, salary, and grade for all employees
* DESCRIBE JOB\_GRADES;
* SELECT e.last\_name AS "Name", e.job\_id AS "Job", d.department\_name AS "Department", e.salary AS "Salary", jg.grade AS "Grade"

FROM employee e

JOIN department d ON e.department\_id = d.department\_id

JOIN job\_grades jg ON e.salary BETWEEN jg.lowest\_sal AND jg.highest\_sal

ORDER BY Grade

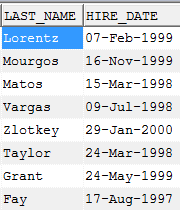


* 1. The HR department wants to determine the names of all employees who were hired after Davies. Create a query to display the name and hire date of any employee hired after employee Davies.

- SELECT last\_name, hire\_date

FROM employee

WHERE hire\_date > (SELECT hire\_date FROM employee WHERE last\_name = 'Davies')



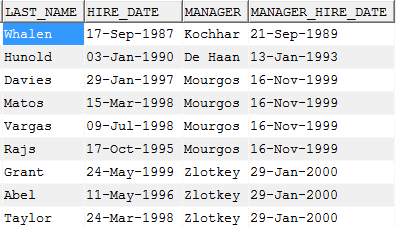
* 1. The HR department needs to find the names and hire dates for all employees who were hired before their managers, along with their managers’ names and hire dates

- SELECT e.last\_name,TO\_CHAR(e.hire\_date,'fmDD Month YYYY') as hire\_date,m.last\_name as Manager,TO\_CHAR(m.hire\_date,'fmDD Month YYYY') as Manager\_hire\_date

FROM employee e

JOIN employee m ON e.manager\_id = m.employee\_id

WHERE e.hire\_date < m.hire\_date



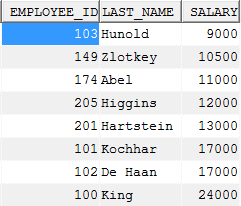
* 1. Create a report that displays the employee number, last name, and salary of all employees who earn more than the average salary. Sort the results in order of ascending salary.

- SELECT employee\_id, last\_name, salary

FROM employee

WHERE salary > (SELECT AVG(salary) FROM employee)

ORDER BY salary ASC;



* 1. Write a query that displays the employee number and last name of all employees who work in a department with any employee whose last name contains a “u”.
* SELECT e.employee\_id, e.last\_name

FROM employees

WHERE e.department\_id IN

(

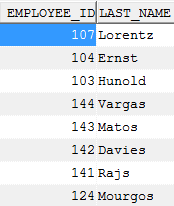
SELECT DISTINCT s.department\_id

FROM employees s

WHERE s.last\_name LIKE '%u%'

)

ORDER BY e.employee\_id;



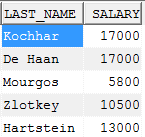
* 1. Create a report for HR that displays the last name and salary of every employee who reports to King.
* SELECT employees.last\_name, employees.salary

FROM employees

INNER JOIN employees manager

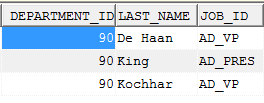
ON employees.manager\_id = manager.employee\_id

WHERE manager.last\_name = 'King';



* 1. Create a report for HR that displays the department number, last name, and job ID for every employee in the Executive department.

SELECT e.department\_id,e.last\_name,e.job\_id

FROM Employee e INNER JOIN department d on e.department\_id = d.department\_id WHERE d.department\_name like 'Executive'

* 1. The HR department needs a report with the following specifications:
  + Last name and department ID of all the employees from the EMPLOYEES table, regardless of whether or not they belong to a department
  + Department ID and department name of all the departments from the DEPARTMENTS table, regardless of whether or not they have employees working in them

Write a compound query to accomplish this.

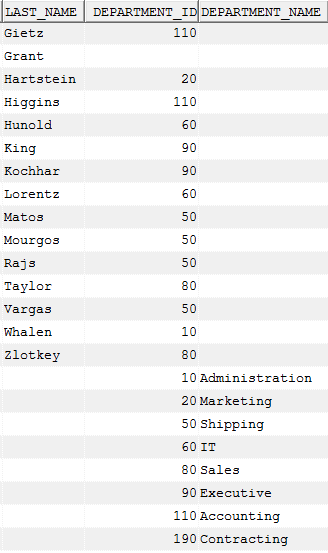
* SELECT last\_name, department\_id

FROM employees

UNION ALL

SELECT department\_id, department\_name

FROM departments;



* 1. Create table My\_Employee from Employees table.
  + Create Insert statement to insert the following row:



* + Change the last name of employee 207 to FPTS

**UPDATE** employee

SET last\_name = 'FPTS'

**WHERE** employee\_id = 207;

* + Confirm your changes to the table.

**SELECT** \* **FROM** employee a **where** a.employee\_id = 207

* + Delete TEST from My\_Employee table

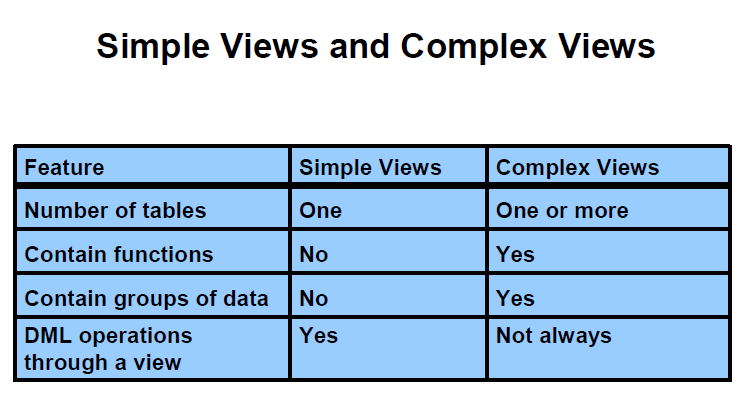
**DELETE** **FROM** employee e

**WHERE** e.employee\_id = 207

* + Commit all pending changes

# CHAPTER 4

1. **Create simple & create complex view**



There are two classifications for views: simple and complex. The basic difference is related to the DML (INSERT, UPDATE, and DELETE) operations.

• A simple view is one that:

- Derives data from only one table

- Contains no functions or groups of data

- Can perform DML operations through the view

• A complex view is one that:

- Derives data from many tables

- Contains functions or groups of data

- Does not always allow DML operations through the view

Syntax:

**CREATE [OR REPLACE] [FORCE|NOFORCE] VIEW *view***

**[(*alias*[, *alias*]...)]**

**AS *subquery***

**[WITH CHECK OPTION [CONSTRAINT *constraint*]]**

**[WITH READ ONLY [CONSTRAINT *constraint*]];**

In the syntax:

OR REPLACE re-creates the view if it already exists

FORCE creates the view regardless of whether or not the base tables exist

NOFORCE creates the view only if the base tables exist (This is the default.)

view is the name of the view

alias specifies names for the expressions selected by the view’s query. (The number of aliases must match the number of expressions selected by the view.)

subquery is a complete SELECT statement (You can use aliases for the

columns in the SELECT list.)

WITH CHECK OPTION specifies that only those rows that are accessible to the view can be inserted or updated

constraint is the name assigned to the CHECK OPTION constraint

WITH READ ONLY ensures that no DML operations can be performed on this view

Example:

* Simple view

**CREATE** **VIEW** empvu80

**AS** **SELECT** employee\_id, last\_name, salary

**FROM** employees

**WHERE** department\_id = 80;

**CREATE** **VIEW** salvu50

**AS** **SELECT** employee\_id ID\_NUMBER, last\_name **NAME**, salary\*12 ANN\_SALARY

**FROM** employees

**WHERE** department\_id = 50;

**CREATE** **OR** **REPLACE** **VIEW** empvu80

(id\_number, **name**, sal, department\_id)

**AS** **SELECT** employee\_id, first\_name || ' ' || last\_name, salary, department\_id

**FROM** employees

**WHERE** department\_id = 80;

\* WITH CHECK OPTION

**CREATE** **OR** **REPLACE** **VIEW** empvu20

**AS** **SELECT** \*

**FROM** employees

**WHERE** department\_id = 20

**WITH** **CHECK** **OPTION** **CONSTRAINT** empvu20\_ck ;

If you run:

**UPDATE** empvu20

SET department\_id = 10

**WHERE** employee\_id = 201;

ERROR at line 1:

ORA-01402: view WITH CHECK OPTION where-clause violation

\* WITH READ ONLY

**CREATE** **OR** **REPLACE** **VIEW** empvu10

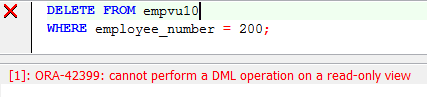
(employee\_number, employee\_name, job\_title)

**AS** **SELECT** employee\_id, last\_name, job\_id

**FROM** employees

**WHERE** department\_id = 10

**WITH** **READ** **ONLY** ;



* Complex view

**CREATE** **OR** **REPLACE** **VIEW** dept\_sum\_vu

(**name**, minsal, maxsal, avgsal)

**AS** **SELECT** d.department\_name, **MIN**(e.salary),

**MAX**(e.salary),**AVG**(e.salary)

**FROM** employees e **JOIN** departments d

**ON** (e.department\_id = d.department\_id)

**GROUP** **BY** d.department\_name;

Note:

\* You can perform DML operations on data through a view if those operations follow certain rules.

You can remove a row from a view unless it contains any of the following:

• Group functions

• A GROUP BY clause

• The DISTINCT keyword

• The pseudocolumn ROWNUM keyword

\* You cannot modify data in a view if it contains:

• Group functions

• A GROUP BY clause

• The DISTINCT keyword

• The pseudocolumn ROWNUM keyword

\* You cannot add data through a view if the view includes:

• Group functions

• A GROUP BY clause

• The DISTINCT keyword

• The pseudocolumn ROWNUM keyword

• Columns defined by expressions

• NOT NULL columns in the base tables that are not selected by the view

1. **Drop view**

Syntax:

**DROP VIEW *view*;**

Example:

**DROP** **VIEW** empvu80;

1. **Using view**

You can retrieve data from a view as you would from any table.

Syntax:

**SELECT \*|{[DISTINCT] *column*|*expression* [*alias*],...}**

**FROM *table;***

Example:

Select \* from empvu80

1. **Practices**
   1. The staff in the HR department wants to hide some of the data in the EMPLOYEES table. They want a view called vw\_EMPLOYEES based on the employee numbers, employee names, and department numbers from the EMPLOYEES table. They want the heading for the employee name to be EMPLOYEE.
   2. Confirm that the view works. Display the contents of the vw\_EMPLOYEES view.
   3. Using your vw\_EMPLOYEES view, write a query for the HR department to display all employee names and department numbers.
   4. Department 50 needs access to its employee data. Create a view named DEPT50 that contains the employee numbers, employee last names, and department numbers for all employees in department 50. You have been asked to label the view columns EMPNO, EMPLOYEE, and DEPTNO. For security purposes, do not allow an employee to be reassigned to another department through the view.
   5. Test your view. Attempt to reassign Matos to department 80.

# CHAPTER 5

1. **Sequence**

* Syntax :

**CREATE SEQUENCE *sequence***

**[INCREMENT BY *n*]**

**[START WITH *n*]**

**[{MAXVALUE *n* | NOMAXVALUE}]**

**[{MINVALUE *n* | NOMINVALUE}]**

**[{CYCLE | NOCYCLE}]**

**[{CACHE *n* | NOCACHE}];**

In the syntax:

*sequence* is the name of the sequence generator

INCREMENT BY *n* specifies the interval between sequence numbers, where *n* is an integer (If this clause is omitted, the sequence increments by 1.)

START WITH *n* specifies the first sequence number to be generated (If this clause is omitted, the sequence starts with 1.)

MAXVALUE *n* specifies the maximum value the sequence can generate

NOMAXVALUE specifies a maximum value of 10^27 for an ascending

sequence and –1 for a descending sequence (This is the default option.)

MINVALUE *n* specifies the minimum sequence value

NOMINVALUE specifies a minimum value of 1 for an ascending sequence and –(10^26) for a descending sequence (This is the default option.)

CYCLE | NOCYCLE specifies whether the sequence continues to generate values after reaching its maximum or minimum value

(NOCYCLE is the default option.)

CACHE n | NOCACHE specifies how many values the Oracle server preallocates and keeps in memory (By default, the Oracle server caches 20 values.)

* Example:

**CREATE** **SEQUENCE** dept\_deptid\_seq

**INCREMENT** **BY** 10

START **WITH** 120

**MAXVALUE** 9999

**NOCACHE**

**NOCYCLE**;

* Using:

NEXTVAL returns the next available sequence value. It returns a unique value every time it is referenced, even for different users.

CURRVAL obtains the current sequence value.

Example:

**INSERT** **INTO** departments(department\_id,department\_name, location\_id)

**VALUES** (dept\_deptid\_seq.**NEXTVAL**,'Support', 2500);

**SELECT** dept\_deptid\_seq.**CURRVAL** **FROM** dual;

* Note:

Rules for Using NEXTVAL and CURRVAL

\* You can use NEXTVAL and CURRVAL in the following contexts:

• The SELECT list of a SELECT statement that is not part of a subquery

• The SELECT list of a subquery in an INSERT statement

• The VALUES clause of an INSERT statement

• The SET clause of an UPDATE statement

\* You cannot use NEXTVAL and CURRVAL in the following contexts:

• The SELECT list of a view

• A SELECT statement with the DISTINCT keyword

• A SELECT statement with GROUP BY, HAVING, or ORDER BY clauses

• A subquery in a SELECT, DELETE, or UPDATE statement

• The DEFAULT expression in a CREATE TABLE or ALTER TABLE statement

\* Gaps in sequence values can occur when:

• A rollback occurs.

• The system crashed.

• A sequence is used in another table.

* Modifying a sequence: Change the increment values, maximum values, minimum value, cycle option or cache option

**ALTER** **SEQUENCE** dept\_deptid\_seq

**INCREMENT** **BY** 20

**MAXVALUE** 999999

**NOCACHE**

**NOCYCLE**;

\* Guidelines for Modifying a Sequence

• You must be the owner or hace the ALTER privilege for the sequence.

• Only future sequence numbers are affected.

• The sequence must be dropped and re\_created to restart the sequence at a different number.

• Some validation is performed.

• To remove a sequence, use the DROP statement:

**DROP** **SEQUENCE** dept\_deptid\_seq

1. **Index**

* Introduction

Is a schema object

• Can be used by the Oracle server to speed up the

retrieval of rows by using a pointer

• Can reduce disk I/O by using a rapid path access

method to locate data quickly

• Is independent of the table that it indexes

• Is used and maintained automatically by the Oracle server

* Types of Indexes

Automatically: A unique index is created automatically when you define a PRIMARY KEY or UNIQUE constraint in a table definition.

Manually: Users can create nonunique indexes on columns to speed up access to the rows.

* Creating an Index

Syntax :

**CREATE INDEX *index***

**ON *table* (*column*[, *column*]...);**

Example

**CREATE** **INDEX** emp\_last\_name\_idx

**ON** employees(last\_name);

* Remove an Index

Syntax:

**DROP INDEX *index*;**

Example:

**DROP** **INDEX** emp\_last\_name\_idx;

1. **Synonyms**

Synonyms are database objects that enable you to call a table by another name. You can create synonyms to give an alternate name to a table.

Simplify access to objects by creating a synonym

(another name for an object). With synonyms, you can:

• Create an easier reference to a table that is owned

by another user

• Shorten lengthy object names

Syntax:

**CREATE [PUBLIC] SYNONYM *synonym***

**FOR *object*;**

In the syntax:

PUBLIC creates a synonym that is accessible to all users

*synonym* is the name of the synonym to be created

*object* identifies the object for which the synonym is created

Example:

**CREATE** **SYNONYM** emp

**FOR** employees;

**select** \* **from** emp;

**DROP** **SYNONYM** emp;

1. **Practices:**
   1. You need a sequence that can be used with the primary key column of the departments table. The sequence should start at 200 and have a maximum value of 1,000. Have your sequence increment by 10

- CREATE SEQUENCE dept\_seq

START WITH 200

INCREMENT BY 10

MAXVALUE 1000

CYCLE;

* 1. To test your sequence, write a script to insert two rows in the departments table

- INSERT INTO department (department\_id, department\_name, manager\_id, location\_id)

VALUES (dept\_seq.NEXTVAL, 'Marketing',dept\_seq.NEXTVAL,2000);

INSERT INTO department (department\_id, department\_name, manager\_id, location\_id)

VALUES (dept\_seq.NEXTVAL, 'IT',dept\_seq.NEXTVAL,1800);

Modify your sequence to have a maximum values of 10,000 and increment by 20.

- ALTER SEQUENCE dept\_seq

MAXVALUE 10000

INCREMENT BY 20;

* 1. Create a nonunique index on the NAME column in the departments table.

- CREATE INDEX dept\_name\_idx

ON departments (name);

* 1. Create a synonym for your EMPLOYEES table. Call it EMP

CREATE SYNONYM emp FOR employees;

# CHAPTER 6

1. **PL/SQL**

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

**Syntax:**

**DECLARE**

<declarations **section**>

**BEGIN**

<executable command(s)>

**EXCEPTION**

<**exception** handling>

**END**;

*Declarations*

This section starts with the keyword DECLARE. It is an optional section and defines all variables, cursors, subprograms, and other elements to be used in the program.

*Executable Commands*

This section is enclosed between the keywords BEGIN and END and it is a mandatory section. It consists of the executable PL/SQL statements of the program. It should have at least one executable line of code, which may be just a NULL command to indicate that nothing should be executed

*Exception Handling*

This section starts with the keyword EXCEPTION. This optional section contains exception(s) that handle errors in the program.

1. **Variables**

* Use of Variables

Variables can be used for:

* + Temporary storage of data
  + Manipulation of stored values
  + Reusability
* Syntax:

***identifier* [CONSTANT] *datatype* [NOT NULL]**

**[:= | DEFAULT *expr*];**

In the syntax:

*identifier* Is the name of the variable

CONSTANT Constrains the variable so that its value cannot change (Constants must be initialized.)

*data type* Is a scalar, composite, reference, or LOB data type (This course covers only scalar, composite, and LOB data types.)

NOT NULL Constrains the variable so that it contains a value (NOT NULL

variables must be initialized.)

*expr* Is any PL/SQL expression that can be a literal expression, another variable, or an expression involving operators and functions

* Variable Name:
  + Must start with a letter
  + Can include letters or numbers
  + Can include special characters (such as $, \_, and #)
  + Must contain no more than 30 characters
  + Must not include reserved words
* Example:

**DECLARE**

v\_myName **VARCHAR2**(20);

**BEGIN**

DBMS\_OUTPUT.PUT\_LINE('My name is: '|| v\_myName);

v\_myName := 'John';

DBMS\_OUTPUT.PUT\_LINE('My name is: '|| v\_myName);

**END**;

**DECLARE**

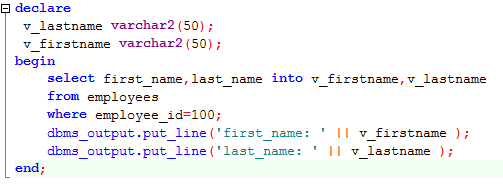
v\_myName **VARCHAR2**(20):= 'John';

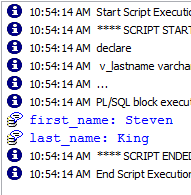
**BEGIN**

v\_myName := 'Steven';

DBMS\_OUTPUT.PUT\_LINE('My name is: '|| v\_myName);

**END**;





\* Basic

**DECLARE**

birthday **DATE**;

emp\_count **SMALLINT** := 0;

\* Constants

**DECLARE**

credit\_limit **CONSTANT** **REAL** := 5000.00;

max\_days\_in\_year **CONSTANT** **INTEGER** := 366;

urban\_legend **CONSTANT** **BOOLEAN** := **FALSE**;

\* Using Default

blood\_type **CHAR** := 'O';

can be rewritten **as** **follows**:

blood\_type **CHAR** **DEFAULT** 'O';

\* Using Not Null

**DECLARE**

acct\_id **INTEGER**(4) **NOT** **NULL** := 9999;

\* Using the %TYPE Attribute

The %TYPE attribute provides the datatype of a variable or database column.

• Is used to declare a variable according to:

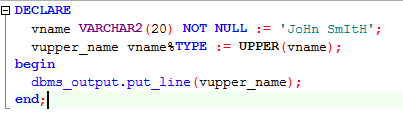
– A database column definition

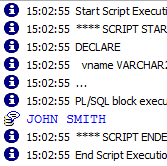
– Another declared variable

• Is prefixed with:

– The database table and column name

– The name of the declared variable





\* Using the %ROWTYPE Attribute

The %ROWTYPE attribute provides a record type that represents a row in a table or view. Columns in a row and corresponding fields in a record have the same names and datatypes. However, fields in a %ROWTYPE record do not inherit constraints, such as the NOT NULL or check constraint, or default values

**DECLARE**

emprec employees%**ROWTYPE**;

**BEGIN**

emprec.EMPLOYEE\_ID := **NULL**; -- this works, null constraint is not inherited

emprec.FIRST\_NAME := 'test';

DBMS\_OUTPUT.PUT\_LINE('emprec.deptname: ' || emprec.FIRST\_NAME);

**END**;

1. **Control Structures**
   1. **Conditional Control**
      1. If statement

Syntax:

**IF *condition* THEN**

***statements*;**

**[ELSIF *condition* THEN**

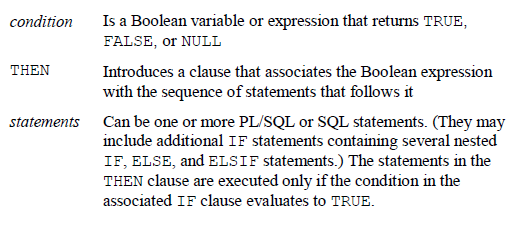
***statements*;]**

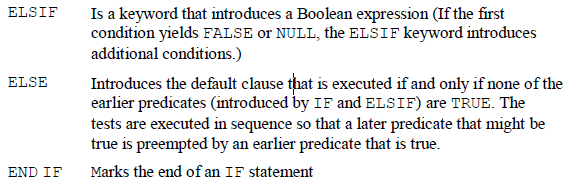
**[ELSE**

***statements*;]**

**END IF;**

In the syntax:





Example:

**DECLARE**

v\_myage **number** :=10;

**BEGIN**

**IF** v\_myage < 11 **THEN**

DBMS\_OUTPUT.PUT\_LINE(' I am a child ');

**ELSE**

DBMS\_OUTPUT.PUT\_LINE(' I am not a child ');

**end** **if**;

**END**;

* + 1. Case when statement

Syntax:

**CASE selector**

**WHEN expression1 THEN result1**

**WHEN expression2 THEN result2**

**...**

**WHEN expressionN THEN resultN**

**[ELSE resultN+1]**

**END;**

**Example:**

**SELECT** last\_name, job\_id, salary,

**CASE** job\_id **WHEN** 'IT\_PROG' **THEN** 1.10\*salary

**WHEN** 'ST\_CLERK' **THEN** 1.15\*salary

**WHEN** 'SA\_REP' **THEN** 1.20\*salary

**ELSE** salary

**END** "REVISED\_SALARY"

**FROM** employees;

**SELECT** last\_name,salary,

(**CASE** **WHEN** salary<5000 **THEN** 'Low'

**WHEN** salary<10000 **THEN** 'Medium'

**WHEN** salary<20000 **THEN** 'Good'

**ELSE** 'Excellent'

**END**) qualified\_salary

**FROM** employees;

* + 1. Decode function

Syntax:

**DECODE(*col|expression, search1, result1***

**[*, search2, result2,...,*][*, default*])**

**Example:**

**SELECT** last\_name, job\_id, salary,

**DECODE**(job\_id, 'IT\_PROG', 1.10\*salary,'ST\_CLERK', 1.15\*salary,

'SA\_REP', 1.20\*salary,salary) REVISED\_SALARY

**FROM** employees;

* 1. Interactive Control
     1. Loop statement

Loops repeat a statement (or a sequence of statements) multiple times.

There are three loop types:

– Basic loop

– FOR loop

– WHILE loop

* Basic loop
  + Syntax:

**LOOP**

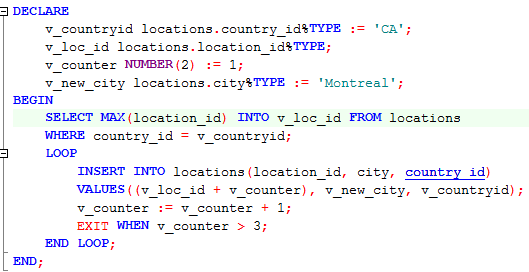
***statement1*;**

**. . .**

**EXIT [WHEN *condition*];**

**END LOOP;**

* + Example



* For loop
  + Syntax

**FOR *counter* IN [REVERSE]**

***lower\_bound..upper\_bound* LOOP**

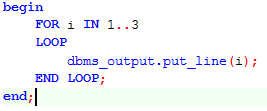
**statement1;**

**statement2;**

**. . .**

**END LOOP;**

* + Example:



* While loop
  + Syntax:

**WHILE *condition* LOOP**

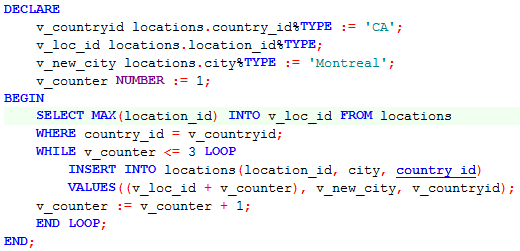
**statement1;**

**statement2;**

**. . .**

**END LOOP;**

* + Example:



* 1. **Sequential Control**
     1. **Goto**

The GOTO statement branches unconditionally to a statement label or block label. The label must be unique within its scope and must precede an executable statement or a PL/SQL block. The GOTO statement transfers control to the labelled statement or block.

Example:

**DECLARE**

p **VARCHAR2**(30);

n **PLS\_INTEGER** := 37; -- test any integer > 2 for prime

**BEGIN**

**FOR** j **in** 2..**ROUND**(**SQRT**(n)) **LOOP**

**IF** n **MOD** j = 0 **THEN** -- test for prime

p := ' is not a prime number'; -- not a prime number

**GOTO** print\_now;

**END** **IF**;

**END** **LOOP**;

p := ' is a prime number';

<<print\_now>>

DBMS\_OUTPUT.PUT\_LINE(**TO\_CHAR**(n) || p);

**END**;

* + 1. **Null statements**

The NULL statement only passes control to the next statement. Some languages refer to such an instruction as a no-op (no operation).

Cursor

A cursor is a pointer to the private memory area allocated by the Oracle Server. It is used to handle the result set of a SELECT statement.

There are two types of cursors: implicit and explicit.

– Implicit: Created and managed internally by the Oracle

Server to process SQL statements

– Explicit: Declared explicitly by the programmer

Example

Implicit

**DECLARE**

l\_vc\_first\_name employees.first\_name%**type**;

l\_vc\_last\_name employees.last\_name%**type**;

**BEGIN**

**SELECT** first\_name, last\_name

**INTO** l\_vc\_first\_name, l\_vc\_last\_name

**FROM** employees

**WHERE** employee\_id=100;

dbms\_output.put\_line('Number of rows processed: '||**sql**%rowcount);

**END**;

**DECLARE**

l\_vc\_first\_name employees.first\_name%**type**;

l\_vc\_last\_name employees.last\_name%**type**;

**BEGIN**

**for** cur **in** (**SELECT** first\_name, last\_name **FROM** employees)

**loop**

dbms\_output.put\_line(cur.first\_name);

**end** **loop**;

**END**;

**DECLARE**

**CURSOR** cur **IS** **SELECT** first\_name, last\_name **FROM** employees;

**BEGIN**

**FOR** loop\_emp **IN** cur

**LOOP**

dbms\_output.put\_line(loop\_emp.first\_name);

dbms\_output.put\_line('Number of rows processed: ||**NVL**(**TO\_CHAR**(cur%rowcount),'Null'));

**END** **LOOP** loop\_emp;

**END**;

Explicit:

**DECLARE**

emp\_rec employees%**rowtype**;

**CURSOR** emp\_cur **IS** **SELECT** \* **FROM** employees;

**BEGIN**

**OPEN** emp\_cur;

**LOOP**

EXIT **WHEN** emp\_cur%**NOTFOUND**;

**FETCH** emp\_cur **INTO** emp\_rec;

dbms\_output.put\_line (emp\_rec.first\_name || ' ' || emp\_rec.last\_name);

**end** **loop**;

**CLOSE** emp\_cur;

**END**;

1. **Handing Exceptions**

An exception is a PL/SQL error that is raised during program execution

Syntax:

**EXCEPTION**

**WHEN *exception1* [OR *exception2* . . .] THEN**

***statement1*;**

***statement2*;**

**. . .**

**[WHEN *exception3* [OR *exception4* . . .] THEN**

***statement1*;**

***statement2*;**

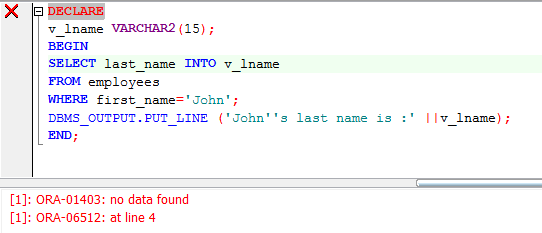
**. . .]**

**[WHEN OTHERS THEN**

***statement1*;**

***statement2*;**

Example:



**DECLARE**

v\_lname **VARCHAR2**(15);

**BEGIN**

**SELECT** last\_name **INTO** v\_lname

**FROM** employees

**WHERE** first\_name='John';

DBMS\_OUTPUT.PUT\_LINE ('John''s last name is :' ||v\_lname);

**EXCEPTION**

**WHEN** NO\_DATA\_FOUND **THEN**

DBMS\_OUTPUT.PUT\_LINE (' Your select statement retrieved 0 row.');

**END**;

**DECLARE**

v\_deptno **NUMBER** := 500;

v\_name **VARCHAR2**(20) := 'Testing';

e\_invalid\_department **EXCEPTION**;

**BEGIN**

**UPDATE** departments

SET department\_name = v\_name

**WHERE** department\_id = v\_deptno;

**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**;

1. **Practices**
   1. The %TYPE attribute:

1. Is used to declare a variable according to a database column definition

2. Is used to declare a variable according to a collection of columns in a database table or view

3. Is used to declare a variable according to the definition of another declared variable

4. Is prefixed with the database table and column name or the name of the declared variable

* 1. Using the DECODE function, write a query that displays the grade of all employees based on the value of the column JOB\_ID, using the following data:

***Job Grade***

AD\_PRES 🡪A

ST\_MAN 🡪 B

IT\_PROG 🡪 C

SA\_REP 🡪 D

ST\_CLERK 🡪 E

None of the above 🡪 0

SELECT last\_name,

DECODE(job\_id, 'AD\_PRES', 'A',

'ST\_MAN', 'B',

'IT\_PROG', 'C',

'SA\_REP', 'D',

'ST\_CLERK', 'E',

'F') AS grade

FROM employees;

* 1. Rewrite the statement in the preceding exercise using the CASE syntax.

SELECT last\_name,

CASE job\_id

WHEN 'AD\_PRES' THEN 'A'

WHEN 'ST\_MAN' THEN 'B'

WHEN 'IT\_PROG' THEN 'C'

WHEN 'SA\_REP' THEN 'D'

WHEN 'ST\_CLERK' THEN 'E'

ELSE 'F'

END AS grade

FROM employees;

* 1. Display the last name, hire date, and day of the week on which the employee started. Label the column DAY. Order the results by the day of the week, starting with Monday.

SELECT last\_name,

hire\_date,

TO\_CHAR(hire\_date, 'DAY') AS day

FROM employees

ORDER BY TO\_CHAR(hire\_date, 'D');



* 1. The HR department needs a report to display the employee number, last name, salary, and salary increased by 15.5% (expressed as a whole number) for each employee. Label the column New Salary

- SELECT employee\_id,

last\_name,

salary,

ROUND(salary \* 1.155, 0) AS "New Salary"

FROM employees;

* 1. Write a query that displays the last name (with the first letter uppercase and all other letters lowercase) and the length of the last name for all employees whose name starts with the letters *J, A,* or *M.*

- SELECT INITCAP(last\_name) AS name,

LENGTH(last\_name) AS length

FROM employees

WHERE last\_name LIKE 'J%'

OR last\_name LIKE 'A%'

OR last\_name LIKE 'M%';

* 1. Create table Emp3 table based on the structure of the Employees table. Use cursor to insert data from Employees to Emp3.

- DECLARE

CURSOR c\_emp IS

SELECT \*

FROM employees;

v\_emp c\_emp%ROWTYPE;

BEGIN

EXECUTE IMMEDIATE 'CREATE TABLE emp3 AS SELECT \* FROM employees WHERE 1=2';

OPEN c\_emp;

LOOP

FETCH c\_emp INTO v\_emp;

EXIT WHEN c\_emp%NOTFOUND;

INSERT INTO emp3 VALUES v\_emp;

END LOOP;

CLOSE c\_emp;

END;

* 1. Write program prints grade of all employees base question b above

-

# CHAPTER 7

1. **Single-row function**

These functions operate on single rows only and return one result per row. There are different types of single-row functions. This lesson covers the following ones:

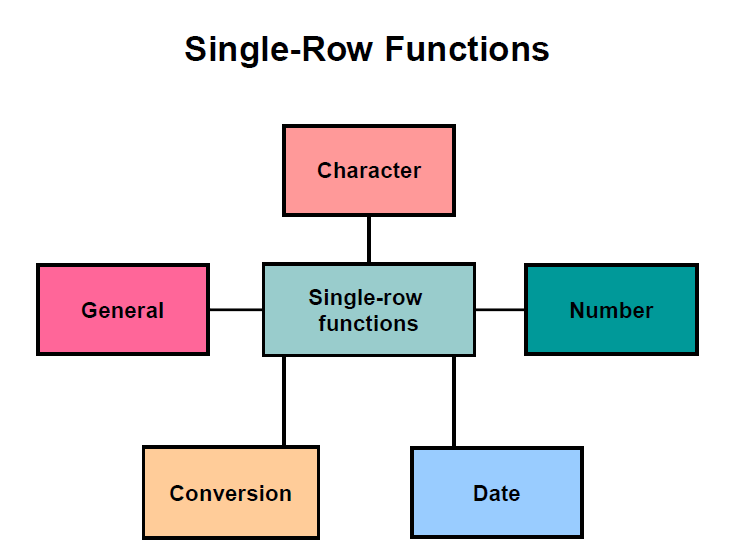
• Character

• Number

• Date

• Conversion

• General



**Character functions**: Accept character input and can return both character and

number values

• **Number functions**: Accept numeric input and return numeric values

• **Date functions**: Operate on values of the DATE data type (All date functions return a value of DATE data type except the MONTHS\_BETWEEN function, which returns a

number.)

• **Conversion functions**: Convert a value from one data type to another

• **General functions:**

- NVL

- NVL2

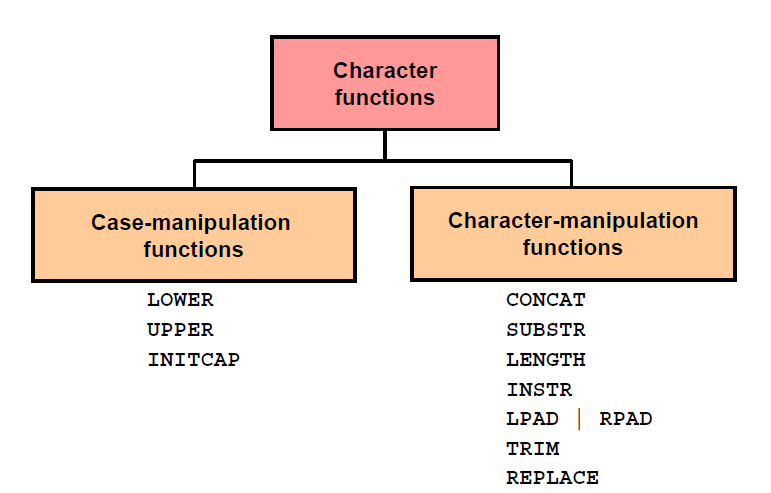
- NULLIF

- COALESCE

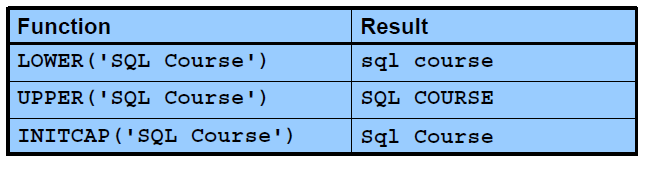
- CASE

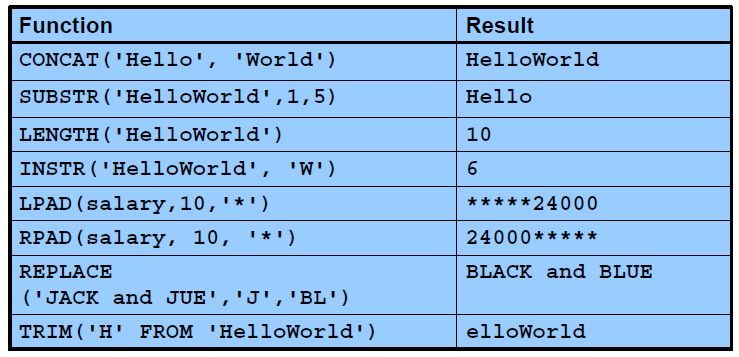
- DECODE

a) Character



Example:





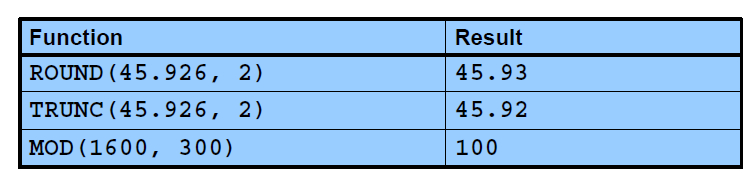
b) Number

ROUND: Rounds value to specified decimal

TRUNC: Truncates value to specified decimal

MOD: Returns remainder of division

Example:



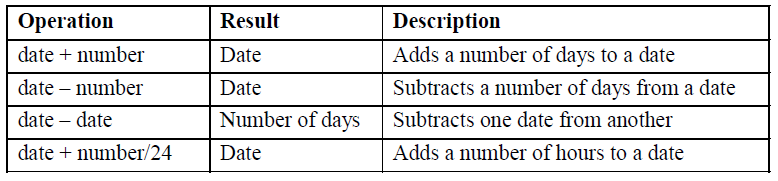
c) Date

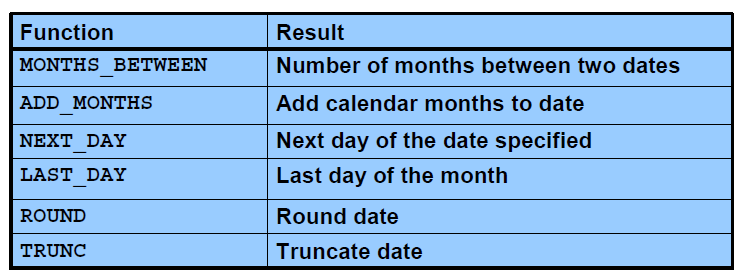
The Oracle database stores dates in an internal numeric format: century, year, month, day, hours, minutes, and seconds.

The default date display format is DD-MON-RR

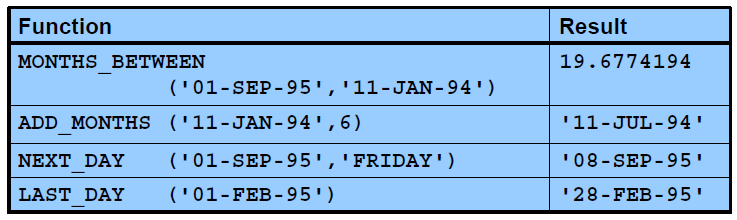
SYSDATE is a function that returns: date and time

You can perform the following operations:

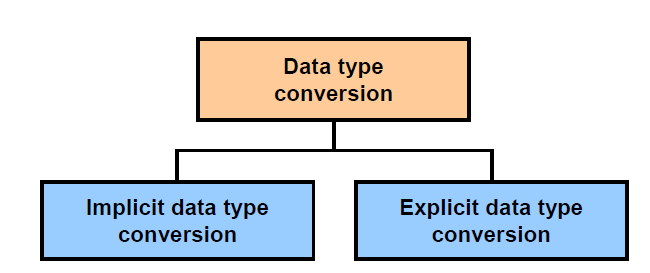




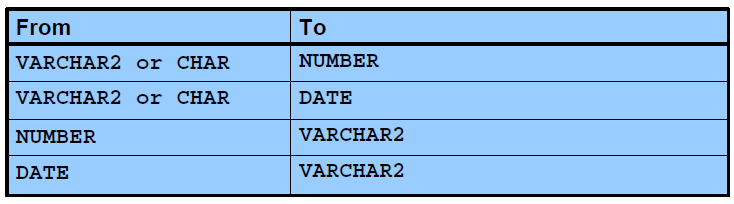
Example:



d) Conversion



* Implicit Data



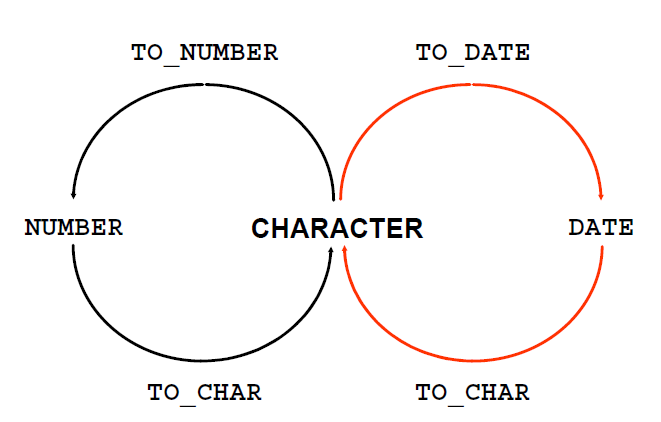
Example:

**select** \* **from** employees **where** hire\_date > '01-JAN-1990'

**select** \* **from** employees **where** salary = 24000

**select** \* **from** employees **where** salary = '24000'

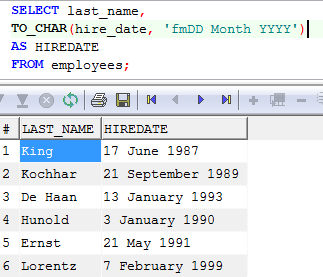
* Explicit Data



Example:

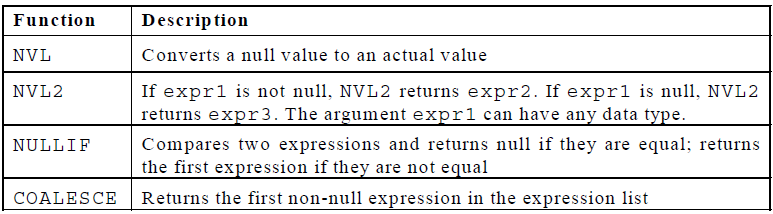
**SELECT** employee\_id, **TO\_CHAR**(hire\_date, 'MM/YY') Month\_Hired

**FROM** employees **WHERE** last\_name = 'Higgins';

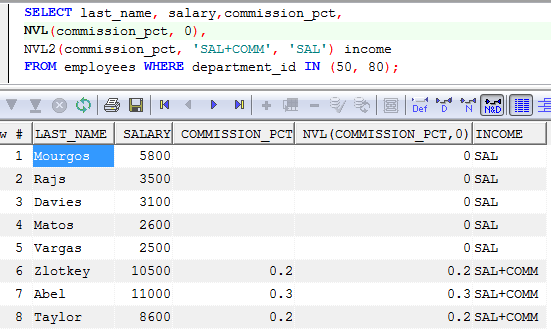


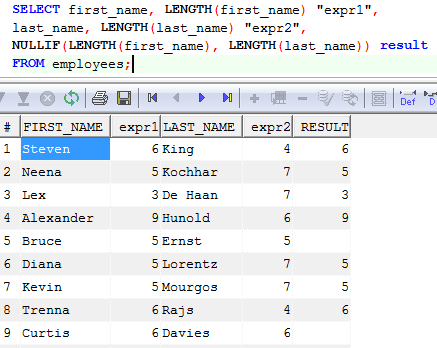
e) General

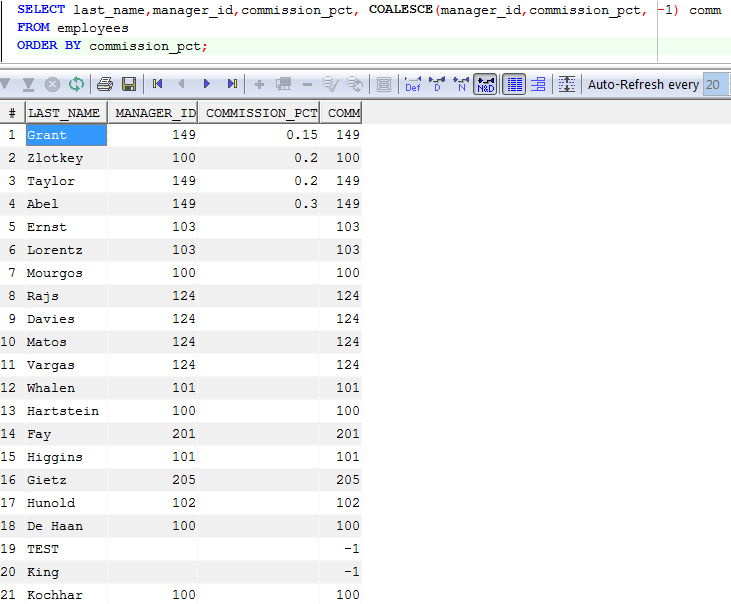
These functions work with any data type and pertain to the use of null values in the expression list.



Example:

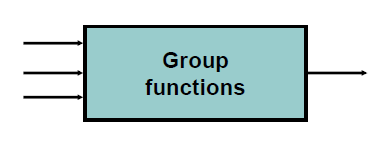






1. **Multiple-row function**

Functions can manipulate groups of rows to give one result per group of rows. These functions are also known as *group functions: Avg, count, max, min, sum*



Syntax:

**SELECT *column*, *group\_function***

**FROM *table***

**[WHERE *condition*]**

**[GROUP BY *group\_by\_expression*]**

**[HAVING *group\_condition*]**

**[ORDER BY *column*];**

In the syntax:

*group\_by\_expression* specifies columns whose values determine the basis for grouping rows

*group\_condition* restricts the groups of rows returned to those groups for which the specified condition is true.

1. **User-defined function**

A **stored function** (also called a **user function** or **user-defined function**) is a set of PL/SQL statements you can call by name. Stored functions are very similar to procedures, except that a function returns a value to the environment in which it is called. User functions can be used as part of a SQL expression

* Syntax:

**CREATE [OR REPLACE] FUNCTION *function\_name***

**[(*argument1* [*mode1*] *datatype1,***

***argument2* [*mode2*] *datatype2,***

***. . .*)]**

**RETURN *datatype***

**IS|AS**

**function\_body;**

**In syntax:**

*function\_name* Is the name of the function to be created

*argument* Is the name given to the function parameter (Every argument is associated with a mode and data type. You can have any number of arguments separated by a comma. You pass the argument when you invoke the function.)

*mode* Is the type of parameter (Only IN parameters should be declared.)

*datatype* Is the data type of the associated parameter

RETURN *datatype* Is the data type of the value returned by the function

*function\_body* Is the PL/SQL block that makes up the function code

* Example

**CREATE** **or** **Replace** **FUNCTION** check\_sal(p\_empno employees.employee\_id%**TYPE**)

**RETURN** **Boolean** **IS**

v\_dept\_id employees.department\_id%**TYPE**;

v\_sal employees.salary%**TYPE**;

v\_avg\_sal employees.salary%**TYPE**;

**BEGIN**

**SELECT** salary,department\_id **INTO** v\_sal,v\_dept\_id **FROM** employees

**WHERE** employee\_id=p\_empno;

**SELECT** **avg**(salary) **INTO** v\_avg\_sal **FROM** employees

**WHERE** department\_id=v\_dept\_id;

**IF** v\_sal > v\_avg\_sal **THEN**

**RETURN** **TRUE**;

**ELSE**

**RETURN** **FALSE**;

**END** **IF**;

**EXCEPTION** **when** others **then** **return** **false**;

**end**;

Invoke

**BEGIN**

DBMS\_OUTPUT.PUT\_LINE('Checking for employee with id 205');

**IF** (check\_sal(205) **IS** **NULL**) **THEN**

DBMS\_OUTPUT.PUT\_LINE('The function returned

NULL due to exception');

**ELSIF** (check\_sal(205)) **THEN**

DBMS\_OUTPUT.PUT\_LINE('Salary > average');

**ELSE**

DBMS\_OUTPUT.PUT\_LINE('Salary < average');

**END** **IF**;

**END**;

1. **Practices**
   1. The HR department wants to find the length of employment for each employee. For each employee, display the last name and calculate the number of months between today and the date on which the employee was hired. Label the column MONTHS\_WORKED. Order your results by the number of months employed. Round the number of months up to the closest whole number.

-SELECT e.last\_name,CEIL(MONTHS\_BETWEEN(SYSDATE,e.hire\_date)) AS Month\_Worked

FROM employee e

ORDER BY Month\_Worked

* 1. Create a report that produces the following for each employee:

<employee last name> earns <salary> monthly but wants <3 times salary>. Label the column Dream Salaries.

- SELECT e.last\_name || ' earns ' || e.salary || ' monthly but wants ' || e.salary\*3 AS Dream

FROM employee e

ORDER BY Dream

* 1. Create a query to display the last name and salary for all employees. Format the salary to be 15 characters long, left-padded with the $ symbol. Label the column SALARY.
  2. Display each employee’s last name, hire date, and salary review date, which is the first Monday after six months of service. Label the column REVIEW. Format the dates to appear in the format similar to “Monday, the Thirty-First of July, 2000.”
  3. Create a query that displays the employees’ last names and commission amounts. If an employee does not earn commission, show “No Commission.” Label the column COMM.
  4. Find the highest, lowest, sum, and average salary of all employees. Label the columns Maximum, Minimum, Sum, and Average, respectively. Round your results to the nearest whole number
  5. Write a query to display the number of people with the same job.
  6. Find the difference between the highest and lowest salaries. Label the column DIFFERENCE.
  7. Create a query to display the total number of employees and, of that total, the number of employees hired in 1995, 1996, 1997, and 1998
  8. Create a function to display all employee of one department.
  9. Write a program that displays titles of employee after 5 year with parameter input employee. Base grades if Grades

D 🡪 TeamLeader.

E 🡪 Manager,

F🡪 Director.

Else Employee.

Know that the employees wage will increase 1 level: after 18-month with Grades = A, B; after 36 months with Grades = C; after 48 months with Grades = D. after 120 months with Grades = E.

# CHAPTER 8

1. **Introduction Stored Procedure**

A **stored procedure** or in simple a **proc** is a named PL/SQL block which performs one or more specific task. This is similar to a procedure in other programming languages.

1. **Create Stored Procedures**

* Syntax:

**CREATE [OR REPLACE] PROCEDURE *procedure\_name***

**[(*argument1* [*mode1*] *datatype1,***

***argument2* [*mode2*] *datatype2,***

***. . .*)]**

**IS|AS**

**procedure\_body;**

In the syntax:

*procedure\_name* Is the name of the procedure to be created

*argument* Is the name given to the procedure parameter. Every argument is associated with a mode and data type. You can have any number of arguments separated by commas.

*mode* Mode of argument:

IN (default)

OUT

IN OUT

*datatype* Is the data type of the associated parameter. The data type of parameters cannot have explicit size; instead, use %TYPE.

*Procedure\_body* Is the PL/SQL block that makes up the code

* *Using sys\_refcursor.*

A REF CURSOR is a PL/SQL data type whose value is the memory address of a query work area on the database. REF\_CURSOR can return a recordset/cursor from a stored procedure or function.

* Example:

**CREATE** **TABLE** dept **AS** **SELECT** \* **FROM** departments;

**CREATE** **PROCEDURE** dept\_add **IS**

v\_dept\_id dept.department\_id%**TYPE**;

v\_dept\_name dept.department\_name%**TYPE**;

**BEGIN**

v\_dept\_id:=280;

v\_dept\_name:='ST-Curriculum';

**INSERT** **INTO** dept(department\_id,department\_name)

**VALUES**(v\_dept\_id,v\_dept\_name);

DBMS\_OUTPUT.PUT\_LINE(' Inserted '|| **SQL**%ROWCOUNT

||' row ');

**END**;

**CREATE** **PROCEDURE** employees\_get(

c\_Emp **OUT** SYS\_REFCURSOR

) **IS**

**BEGIN**

**OPEN** c\_Emp **FOR**

**SELECT** \*

**FROM** employees a

**ORDER** **BY** a.first\_name ;

**END**;

1. **Invoking**

* Syntax:

***BEGIN***

***…***

***procedure\_name (***

***argument1 [mode1] datatype1,***

***argument2 [mode2] datatype2,***

***. . .***

***)***

***…***

***END;***

* Example

1. **Practices**
   1. Create stored procedure to insert data into Employees table with input parameters is all columns, return a message to indicate whether the operation succeeded or not.
   2. Create stored procedure to update Phone\_Number of any Employee with input parameter is Employee\_ID column, return a message to indicate whether the operation succeeded or not.
   3. Create stored procedure to insert data into Departments table with input parameters is all columns, return a message to indicate whether the operation succeeded or not.
   4. Create store procedure to update data Departments table with input parameters is columns, return a message to indicate whether the operation succeeded or not.
   5. Create stored procedure to insert data into Job\_Grades table with input parameters is all columns, return a message to indicate whether the operation succeeded or not.
   6. Create stored procedure to delete data into Job\_Grades table with input parameters is Grade column, return a message to indicate whether the operation succeeded or not.
   7. The HR department needs a report of all employees. Write a stored procedure to get the last name, department number, and department name for all employees.
   8. The HR department needs a report that displays the last name and hire date for all employees who were hired in 1994. Write a stored procedure to get data with input parameter is year.

# CHAPTER 9

1. **GRANT statement**

A user automatically has all object privileges for schema objects contained in the user’s schema. A user can grant any object privilege on any schema object.

* Introduction

Use the GRANT statement to give privileges to a specific user or role, or to all users, to perform actions on database objects. You can also use the GRANT statement to grant a role to a user, to PUBLIC, or to another role.

The following types of privileges can be granted:

* + Delete data from a specific table.
  + Insert data into a specific table.
  + Create a foreign key reference to the named table or to a subset of columns from a table.
  + Select data from a table, view, or a subset of columns in a table.
  + Create a trigger on a table.
  + Update data in a table or in a subset of columns in a table.
  + Run a specified function or procedure.
  + Use a sequence generator or a user-defined type.
* Syntax:

\* Syntax for tables

**GRANT** [privilege-type](https://docs.oracle.com/javadb/10.8.3.0/ref/rrefsqljgrant.html#rrefsqljgrant__grantprivtype) **ON [TABLE] {** [table-Name](https://docs.oracle.com/javadb/10.8.3.0/ref/rreftablename.html#rreftablename) **|** [view-Name](https://docs.oracle.com/javadb/10.8.3.0/ref/rrefviewname.html#rrefviewname) **} TO** [grantees](https://docs.oracle.com/javadb/10.8.3.0/ref/rrefsqljgrant.html#rrefsqljgrant__grantgrantees)

\* Syntax for routines

**GRANT EXECUTE ON { FUNCTION | PROCEDURE }** [routine-designator](https://docs.oracle.com/javadb/10.8.3.0/ref/rrefsqljgrant.html#rrefsqljgrant__grantroutinename) **TO** [grantees](https://docs.oracle.com/javadb/10.8.3.0/ref/rrefsqljgrant.html#rrefsqljgrant__grantgrantees)

* Example

**GRANT** **UPDATE**, **TRIGGER** **ON** **TABLE** t **TO** example

**GRANT** EXECUTE **ON** **PROCEDURE** p **TO** example

1. **Revoke**

* Introduction

Use the REVOKE statement to remove privileges from a specific user or role, or from all users, to perform actions on database objects. You can also use the REVOKE statement to revoke a role from a user, from PUBLIC, or from another role

The following types of privileges can be revoked:

* + Delete data from a specific table.
  + Insert data into a specific table.
  + Create a foreign key reference to the named table or to a subset of columns from a table.
  + Select data from a table, view, or a subset of columns in a table.
  + Create a trigger on a table.
  + Update data in a table or in a subset of columns in a table.
  + Run a specified routine (function or procedure).
  + Use a sequence generator or a user-defined type.
* Syntax:

\* Syntax for tables

**REVOKE** [privilege-type](https://docs.oracle.com/javadb/10.8.3.0/ref/rrefsqljrevoke.html#rrefsqljrevoke__revokeprivtype) **ON [ TABLE ] {** [table-Name](https://docs.oracle.com/javadb/10.8.3.0/ref/rreftablename.html#rreftablename) **|** [view-Name](https://docs.oracle.com/javadb/10.8.3.0/ref/rrefviewname.html#rrefviewname) **} FROM** [grantees](https://docs.oracle.com/javadb/10.8.3.0/ref/rrefsqljrevoke.html#rrefsqljrevoke__revokegrantees)

\* Syntax for routines

**REVOKE EXECUTE ON { FUNCTION | PROCEDURE }** [routine-designator](https://docs.oracle.com/javadb/10.8.3.0/ref/rrefsqljrevoke.html#rrefsqljrevoke__revokeroutinename) **FROM** [grantees](https://docs.oracle.com/javadb/10.8.3.0/ref/rrefsqljrevoke.html#rrefsqljrevoke__revokegrantees) **RESTRICT**

The RESTRICT clause specifies that the EXECUTE privilege cannot be revoked if the specified routine is used in a view, trigger, or constraint, and the privilege is being revoked from the owner of the view, trigger, or constraint.

* Example:

**REVOKE** **UPDATE**, **TRIGGER** **ON** **TABLE** t **FROM** example

**REVOKE** EXECUTE **ON** **PROCEDURE** p **FROM** example RESTRICT

1. **Job**

Job is used to run action at a fixed time

* Create job

begin  
dbms\_scheduler.create\_job (  
   job\_name           =>  'run\_load\_sales',  
   job\_type           =>  'STORED\_PROCEDURE',  
   job\_action         =>  'name of stored procedure',  
   start\_date         =>  '01-MAR-2010 03:00:00 AM',  
   repeat\_interval    =>  'FREQ=DAILY',  
   enabled            =>  TRUE);  
END;

There are a large number of expressions that can be used to define the repeat interval for a job.  Here are a few examples:

|  |  |
| --- | --- |
| **repeat\_interval** | **Description** |
| freq=hourly | Run every hour |
| freq=daily; byhour=3 | Run at 3 am every day |
| freq=daily; byhour=8,20 | Run at 8 am and 8 pm every day |
| freq=monthly; bymonthday=1 | Run on the first day of every month |
| freq=monthly; bymonthday=-1 | Run on the last day of every month |
| freq=yearly; bymonth=sep; bymonthday=20; | Run yearly on September 20th |

Example:

**BEGIN**

dbms\_scheduler.create\_job('"JOB\_RPT\_ASSET\_DAILY"',

job\_type=>'STORED\_PROCEDURE',

job\_action=> 'sprpt\_asset\_daily',

number\_of\_arguments=>0,

start\_date=>**TO\_TIMESTAMP\_TZ**('22-DEC-2017 05.38.49.000000000 PM +07:00','DD-MON-RRRR HH.MI.SSXFF AM TZR','NLS\_DATE\_LANGUAGE=english'),

repeat\_interval=>'FREQ=DAILY; INTERVAL=1; BYDAY=MON,TUE,WED,THU,FRI; BYHOUR=19; BYMINUTE=50',

end\_date=>**NULL**,

job\_class=>'"DEFAULT\_JOB\_CLASS"',

enabled=>**FALSE**,

auto\_drop=>**FALSE**,

comments=>'Tu dong bao cao tai san tung khach hang cho ezmargin, ezreport lay bao cao'

);

dbms\_scheduler.enable('"JOB\_RPT\_ASSET\_DAILY"');

**COMMIT**;

**END**;

**BEGIN**

dbms\_scheduler.create\_job('"JOB\_RPT\_STOCK\_STATEMENT\_I"',

job\_type=>'PLSQL\_BLOCK',

job\_action=>'begin sptrpt\_stock\_statement\_i(1); end;',

number\_of\_arguments=>0,

start\_date=>**TO\_TIMESTAMP\_TZ**('29-NOV-2016 10.43.30.000000000 AM +07:00','DD-MON-RRRR HH.MI.SSXFF AM TZR','NLS\_DATE\_LANGUAGE=english'),

repeat\_interval=>

'FREQ=DAILY; INTERVAL=1; BYDAY=MON,TUE,WED,THU,FRI; BYHOUR=15,18; BYMINUTE=30',

end\_date=>**NULL**,

job\_class=>'"DEFAULT\_JOB\_CLASS"',

enabled=>**FALSE**,

auto\_drop=>**FALSE**,

comments=> 'Sao ke CK daily, chay vao 15h30, 18h30'

);

dbms\_scheduler.enable('"JOB\_RPT\_STOCK\_STATEMENT\_I"');

**COMMIT**;

**END**;

* Running a Job Manually

begin  
dbms\_scheduler.run\_job (job\_name => 'name');  
end;

* Stopping Running Jobs

dbms\_scheduler.stop\_job (job\_name => 'name');  
end;

* Disabling and Enabling Jobs

begin  
dbms\_scheduler.disable (job\_name => 'name',..);  
end;  
To re-enable a job which has been disabled, use the enable procedure.

begin  
dbms\_scheduler.enable (job\_name => 'name',..);  
end;

* Dropping Jobs

begin  
dbms\_scheduler.drop\_job ('name',..);  
end;

1. **Practices**
   1. Create tblLog table with following structures:



* + 1. Write stored procedure “sp\_tblLog\_insert” to insert system date to tblLog table
    2. Create job to run sp\_tblLog\_insert on 6 am on every day. Run job
  1. Grant another user access to your job\_grades table. Take back the privilege from the user.
  2. Grant another user query privilege on your table. Then, verify whether that user can utilize the privilege. There are three users: user01, user02, user03.

User01 contains: Employees

* + Grant user02 privilege to view records in Employees table. Include an option for this user to further grant this privilege to other users.
  + From user02, write to query Employees table of user01
  + Grant privilege to view records in Employees table to a third user
  + Take back the privilege from the user who performs step 1 (user02)
  1. Grant another user query and data manipulation privileges on your Departments table. Make sure that the user cannot pass on these privileges to other users.
  2. Take back the privileges on the Departments table granted to another user.