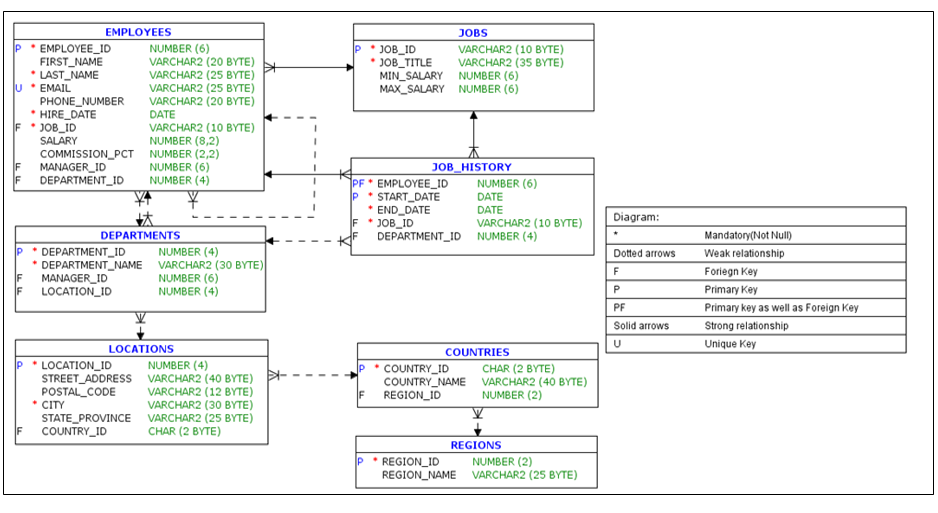
Based on the specifications, the following tables were designed and used for the EMS.



DDL:

ALTER TABLE REGIONS

DROP PRIMARY KEY CASCADE;

DROP TABLE REGIONS CASCADE CONSTRAINTS;

CREATE TABLE REGIONS

(

REGION\_ID NUMBER(2) CONSTRAINT REGION\_ID\_NN NOT NULL,

REGION\_NAME VARCHAR2(25 BYTE)

);

CREATE UNIQUE INDEX REG\_ID\_PK ON REGIONS

(REGION\_ID);

ALTER TABLE REGIONS ADD (

CONSTRAINT REG\_ID\_PK

PRIMARY KEY

(REGION\_ID)

USING INDEX REG\_ID\_PK

ENABLE VALIDATE);

/

ALTER TABLE COUNTRIES

DROP PRIMARY KEY CASCADE;

DROP TABLE COUNTRIES CASCADE CONSTRAINTS;

CREATE TABLE COUNTRIES

(

COUNTRY\_ID CHAR(2 BYTE) CONSTRAINT COUNTRY\_ID\_NN NOT NULL,

COUNTRY\_NAME VARCHAR2(40 BYTE),

REGION\_ID NUMBER(2)

);

CREATE UNIQUE INDEX COUNTRY\_C\_ID\_PK ON COUNTRIES

(COUNTRY\_ID);

ALTER TABLE COUNTRIES ADD (

CONSTRAINT COUNTRY\_C\_ID\_PK

PRIMARY KEY

(COUNTRY\_ID)

USING INDEX COUNTRY\_C\_ID\_PK

ENABLE VALIDATE);

ALTER TABLE COUNTRIES ADD (

CONSTRAINT COUNTR\_REG\_FK

FOREIGN KEY (REGION\_ID)

REFERENCES REGIONS (REGION\_ID)

ENABLE VALIDATE);

/

ALTER TABLE LOCATIONS

DROP PRIMARY KEY CASCADE;

DROP TABLE LOCATIONS CASCADE CONSTRAINTS;

CREATE TABLE LOCATIONS

(

LOCATION\_ID NUMBER(4),

STREET\_ADDRESS VARCHAR2(40 BYTE),

POSTAL\_CODE VARCHAR2(12 BYTE),

CITY VARCHAR2(30 BYTE) CONSTRAINT LOC\_CITY\_NN NOT NULL,

STATE\_PROVINCE VARCHAR2(25 BYTE),

COUNTRY\_ID CHAR(2 BYTE)

);

CREATE INDEX LOC\_CITY\_IX ON LOCATIONS

(CITY);

CREATE INDEX LOC\_COUNTRY\_IX ON LOCATIONS

(COUNTRY\_ID);

CREATE UNIQUE INDEX LOC\_ID\_PK ON LOCATIONS

(LOCATION\_ID);

CREATE INDEX LOC\_STATE\_PROVINCE\_IX ON LOCATIONS

(STATE\_PROVINCE);

ALTER TABLE LOCATIONS ADD (

CONSTRAINT LOC\_ID\_PK

PRIMARY KEY

(LOCATION\_ID)

USING INDEX LOC\_ID\_PK

ENABLE VALIDATE);

ALTER TABLE LOCATIONS ADD (

CONSTRAINT LOC\_C\_ID\_FK

FOREIGN KEY (COUNTRY\_ID)

REFERENCES COUNTRIES (COUNTRY\_ID)

ENABLE VALIDATE);

/

ALTER TABLE JOBS

DROP PRIMARY KEY CASCADE;

DROP TABLE JOBS CASCADE CONSTRAINTS;

CREATE TABLE JOBS

(

JOB\_ID VARCHAR2(10 BYTE),

JOB\_TITLE VARCHAR2(35 BYTE) CONSTRAINT JOB\_TITLE\_NN NOT NULL,

MIN\_SALARY NUMBER(6),

MAX\_SALARY NUMBER(6)

);

CREATE INDEX JOB\_ID\_PK ON JOBS

(JOB\_ID);

ALTER TABLE JOBS ADD (

CONSTRAINT JOB\_ID\_PK

PRIMARY KEY

(JOB\_ID)

USING INDEX JOB\_ID\_PK

ENABLE VALIDATE);

/

ALTER TABLE DEPARTMENTS

DROP PRIMARY KEY CASCADE;

DROP TABLE DEPARTMENTS CASCADE CONSTRAINTS;

CREATE TABLE DEPARTMENTS

(

DEPARTMENT\_ID NUMBER(4),

DEPARTMENT\_NAME VARCHAR2(30 BYTE) CONSTRAINT DEPT\_NAME\_NN NOT NULL,

MANAGER\_ID NUMBER(6),

LOCATION\_ID NUMBER(4)

);

CREATE UNIQUE INDEX DEPT\_ID\_PK ON DEPARTMENTS

(DEPARTMENT\_ID);

ALTER TABLE DEPARTMENTS ADD (

CONSTRAINT DEPT\_ID\_PK

PRIMARY KEY

(DEPARTMENT\_ID)

USING INDEX DEPT\_ID\_PK

ENABLE VALIDATE);

/

ALTER TABLE EMPLOYEES

DROP PRIMARY KEY CASCADE;

DROP TABLE EMPLOYEES CASCADE CONSTRAINTS;

CREATE TABLE EMPLOYEES

(

EMPLOYEE\_ID NUMBER(6),

FIRST\_NAME VARCHAR2(20 BYTE),

LAST\_NAME VARCHAR2(25 BYTE) CONSTRAINT EMP\_LAST\_NAME\_NN NOT NULL,

EMAIL VARCHAR2(25 BYTE) CONSTRAINT EMP\_EMAIL\_NN NOT NULL,

PHONE\_NUMBER VARCHAR2(20 BYTE),

HIRE\_DATE DATE CONSTRAINT EMP\_HIRE\_DATE\_NN NOT NULL,

JOB\_ID VARCHAR2(10 BYTE) CONSTRAINT EMP\_JOB\_NN NOT NULL,

SALARY NUMBER(8,2),

COMMISSION\_PCT NUMBER(2,2),

MANAGER\_ID NUMBER(6),

DEPARTMENT\_ID NUMBER(4)

);

CREATE INDEX EMP\_DEPARTMENT\_IX ON EMPLOYEES

(DEPARTMENT\_ID);

CREATE UNIQUE INDEX EMP\_EMAIL\_UK ON EMPLOYEES

(EMAIL);

CREATE UNIQUE INDEX EMP\_EMP\_ID\_PK ON EMPLOYEES

(EMPLOYEE\_ID);

CREATE INDEX EMP\_JOB\_IX ON EMPLOYEES

(JOB\_ID);

CREATE INDEX EMP\_MANAGER\_IX ON EMPLOYEES

(MANAGER\_ID);

CREATE INDEX EMP\_NAME\_IX ON EMPLOYEES

(LAST\_NAME, FIRST\_NAME);

ALTER TABLE EMPLOYEES ADD (

CONSTRAINT EMP\_SALARY\_MIN

CHECK (salary > 0)

ENABLE VALIDATE,

CONSTRAINT EMP\_EMP\_ID\_PK

PRIMARY KEY

(EMPLOYEE\_ID)

USING INDEX EMP\_EMP\_ID\_PK

ENABLE VALIDATE,

CONSTRAINT EMP\_EMAIL\_UK

UNIQUE (EMAIL)

USING INDEX EMP\_EMAIL\_UK

ENABLE VALIDATE);

ALTER TABLE DEPARTMENTS ADD (

CONSTRAINT DEPT\_LOC\_FK

FOREIGN KEY (LOCATION\_ID)

REFERENCES LOCATIONS (LOCATION\_ID)

ENABLE VALIDATE,

CONSTRAINT DEPT\_MGR\_FK

FOREIGN KEY (MANAGER\_ID)

REFERENCES EMPLOYEES (EMPLOYEE\_ID)

ENABLE VALIDATE);

ALTER TABLE EMPLOYEES ADD (

CONSTRAINT EMP\_DEPT\_FK

FOREIGN KEY (DEPARTMENT\_ID)

REFERENCES DEPARTMENTS (DEPARTMENT\_ID)

ENABLE VALIDATE,

CONSTRAINT EMP\_JOB\_FK

FOREIGN KEY (JOB\_ID)

REFERENCES JOBS (JOB\_ID)

ENABLE VALIDATE,

CONSTRAINT EMP\_MANAGER\_FK

FOREIGN KEY (MANAGER\_ID)

REFERENCES EMPLOYEES (EMPLOYEE\_ID)

ENABLE VALIDATE);

/

ALTER TABLE JOB\_HISTORY

DROP PRIMARY KEY CASCADE;

DROP TABLE JOB\_HISTORY CASCADE CONSTRAINTS;

CREATE TABLE JOB\_HISTORY

(

EMPLOYEE\_ID NUMBER(6) CONSTRAINT JHIST\_EMPLOYEE\_NN NOT NULL,

START\_DATE DATE,

END\_DATE DATE,

JOB\_ID VARCHAR2(10 BYTE) CONSTRAINT JHIST\_JOB\_NN NOT NULL,

DEPARTMENT\_ID NUMBER(4)

);

CREATE INDEX JHIST\_DEPARTMENT\_IX ON JOB\_HISTORY

(DEPARTMENT\_ID);

CREATE INDEX JHIST\_EMPLOYEE\_IX ON JOB\_HISTORY

(EMPLOYEE\_ID);

CREATE UNIQUE INDEX JHIST\_EMP\_ID\_ST\_DATE\_PK ON JOB\_HISTORY

(EMPLOYEE\_ID, START\_DATE);

CREATE INDEX JHIST\_JOB\_IX ON JOB\_HISTORY

(JOB\_ID);

DML:

SET DEFINE OFF;

Insert into REGIONS

(REGION\_ID, REGION\_NAME)

Values

(1, 'Europe');

Insert into REGIONS

(REGION\_ID, REGION\_NAME)

Values

(2, 'Americas');

Insert into REGIONS

(REGION\_ID, REGION\_NAME)

Values

(3, 'Asia');

Insert into REGIONS

(REGION\_ID, REGION\_NAME)

Values

(4, 'Middle East and Africa');

COMMIT;

SET DEFINE OFF;

Insert into COUNTRIES

(COUNTRY\_ID, COUNTRY\_NAME, REGION\_ID)

Values

('AR', 'Argentina', 2);

Insert into COUNTRIES

(COUNTRY\_ID, COUNTRY\_NAME, REGION\_ID)

Values

('AU', 'Australia', 3);

Insert into COUNTRIES

(COUNTRY\_ID, COUNTRY\_NAME, REGION\_ID)

Values

('BE', 'Belgium', 1);

Insert into COUNTRIES

(COUNTRY\_ID, COUNTRY\_NAME, REGION\_ID)

Values

('BR', 'Brazil', 2);

Insert into COUNTRIES

(COUNTRY\_ID, COUNTRY\_NAME, REGION\_ID)

Values

('CA', 'Canada', 2);

Insert into COUNTRIES

(COUNTRY\_ID, COUNTRY\_NAME, REGION\_ID)

Values

('CH', 'Switzerland', 1);

Insert into COUNTRIES

(COUNTRY\_ID, COUNTRY\_NAME, REGION\_ID)

Values

('CN', 'China', 3);

Insert into COUNTRIES

(COUNTRY\_ID, COUNTRY\_NAME, REGION\_ID)

Values

('DE', 'Germany', 1);

Insert into COUNTRIES

(COUNTRY\_ID, COUNTRY\_NAME, REGION\_ID)

Values

('DK', 'Denmark', 1);

Insert into COUNTRIES

(COUNTRY\_ID, COUNTRY\_NAME, REGION\_ID)

Values

('IN', 'INDIA', 3);

COMMIT;

SET DEFINE OFF;

Insert into LOCATIONS

(LOCATION\_ID, STREET\_ADDRESS, POSTAL\_CODE, CITY, STATE\_PROVINCE,

COUNTRY\_ID)

Values

(1800, '147 Spadina Ave', 'M5V 2L7', 'Toronto', 'Ontario',

'CA');

Insert into LOCATIONS

(LOCATION\_ID, STREET\_ADDRESS, POSTAL\_CODE, CITY, STATE\_PROVINCE,

COUNTRY\_ID)

Values

(1700, '6092 Boxwood St', 'YSW 9T2', 'Whitehorse', 'Yukon',

'CA');

Insert into LOCATIONS

(LOCATION\_ID, STREET\_ADDRESS, POSTAL\_CODE, CITY, COUNTRY\_ID)

Values

(1400, '40-5-12 Laogianggen', '190518', 'Beijing', 'CN');

Insert into LOCATIONS

(LOCATION\_ID, STREET\_ADDRESS, POSTAL\_CODE, CITY, STATE\_PROVINCE,

COUNTRY\_ID)

Values

(1500, '1298 Vileparle (E)', '490231', 'Bombay', 'Maharashtra',

'IN');

Insert into LOCATIONS

(LOCATION\_ID, STREET\_ADDRESS, POSTAL\_CODE, CITY, STATE\_PROVINCE,

COUNTRY\_ID)

Values

(2200, '12-98 Victoria Street', '2901', 'Sydney', 'New South Wales',

'AU');

Insert into LOCATIONS

(LOCATION\_ID, STREET\_ADDRESS, POSTAL\_CODE, CITY, STATE\_PROVINCE,

COUNTRY\_ID)

Values

(2700, 'Schwanthalerstr. 7031', '80925', 'Munich', 'Bavaria',

'DE');

Insert into LOCATIONS

(LOCATION\_ID, STREET\_ADDRESS, POSTAL\_CODE, CITY, STATE\_PROVINCE,

COUNTRY\_ID)

Values

(2800, 'Rua Frei Caneca 1360 ', '01307-002', 'Sao Paulo', 'Sao Paulo',

'BR');

Insert into LOCATIONS

(LOCATION\_ID, STREET\_ADDRESS, POSTAL\_CODE, CITY, STATE\_PROVINCE,

COUNTRY\_ID)

Values

(2900, '20 Rue des Corps-Saints', '1730', 'Geneva', 'Geneve',

'CH');

Insert into LOCATIONS

(LOCATION\_ID, STREET\_ADDRESS, POSTAL\_CODE, CITY, STATE\_PROVINCE,

COUNTRY\_ID)

Values

(3000, 'Murtenstrasse 921', '3095', 'Bern', 'BE',

'CH');

COMMIT;

SET DEFINE OFF;

Insert into JOBS

(JOB\_ID, JOB\_TITLE, MIN\_SALARY, MAX\_SALARY)

Values

('AD\_PRES', 'President', 20000, 40000);

Insert into JOBS

(JOB\_ID, JOB\_TITLE, MIN\_SALARY, MAX\_SALARY)

Values

('FI\_MGR', 'Finance Manager', 8200, 16000);

Insert into JOBS

(JOB\_ID, JOB\_TITLE, MIN\_SALARY, MAX\_SALARY)

Values

('FI\_ACCOUNT', 'Accountant', 4200, 9000);

Insert into JOBS

(JOB\_ID, JOB\_TITLE, MIN\_SALARY, MAX\_SALARY)

Values

('PU\_MAN', 'Purchasing Manager', 8000, 15000);

Insert into JOBS

(JOB\_ID, JOB\_TITLE, MIN\_SALARY, MAX\_SALARY)

Values

('ST\_CLERK', 'Stock Clerk', 2000, 5000);

Insert into JOBS

(JOB\_ID, JOB\_TITLE, MIN\_SALARY, MAX\_SALARY)

Values

('IT\_PROG', 'Programmer', 4000, 10000);

Insert into JOBS

(JOB\_ID, JOB\_TITLE, MIN\_SALARY, MAX\_SALARY)

Values

('MK\_REP', 'Marketing Representative', 4000, 9000);

Insert into JOBS

(JOB\_ID, JOB\_TITLE, MIN\_SALARY, MAX\_SALARY)

Values

('PR\_REP', 'Public Relations Representative', 4500, 10500);

Insert into JOBS

(JOB\_ID, JOB\_TITLE, MIN\_SALARY, MAX\_SALARY)

Values

('AD\_VP','Administration Vice President',15000,30000);

Insert into JOBS

(JOB\_ID, JOB\_TITLE, MIN\_SALARY, MAX\_SALARY)

Values

('ST\_MAN','Stock Manager',5500,8500);

Insert into JOBS

(JOB\_ID, JOB\_TITLE, MIN\_SALARY, MAX\_SALARY)

Values

('MK\_MAN','Marketing Manager',9000,15000);

Insert into JOBS

(JOB\_ID, JOB\_TITLE, MIN\_SALARY, MAX\_SALARY)

Values

('AC\_MGR','Accounting Manager',8200,16000);

Insert into JOBS

(JOB\_ID, JOB\_TITLE, MIN\_SALARY, MAX\_SALARY)

Values

('AD\_ASST','Administration Assistant',3000,6000);

COMMIT;

SET DEFINE OFF;

Insert into DEPARTMENTS

(DEPARTMENT\_ID, DEPARTMENT\_NAME, MANAGER\_ID, LOCATION\_ID)

Values

('10', 'Administration', null, '1700');

Insert into DEPARTMENTS

(DEPARTMENT\_ID, DEPARTMENT\_NAME, MANAGER\_ID, LOCATION\_ID)

Values

('20', 'Marketing', null, '1800');

Insert into DEPARTMENTS

(DEPARTMENT\_ID, DEPARTMENT\_NAME, MANAGER\_ID, LOCATION\_ID)

Values

('30', 'Purchasing', null, '1700');

Insert into DEPARTMENTS

(DEPARTMENT\_ID, DEPARTMENT\_NAME, MANAGER\_ID, LOCATION\_ID)

Values

('50', 'Shipping', null, '1500');

Insert into DEPARTMENTS

(DEPARTMENT\_ID, DEPARTMENT\_NAME, MANAGER\_ID, LOCATION\_ID)

Values

('60', 'IT', null, '1400');

Insert into DEPARTMENTS

(DEPARTMENT\_ID, DEPARTMENT\_NAME, MANAGER\_ID, LOCATION\_ID)

Values

('70', 'Public Relations', null, '2700');

Insert into DEPARTMENTS

(DEPARTMENT\_ID, DEPARTMENT\_NAME, MANAGER\_ID, LOCATION\_ID)

Values

('90', 'Executive', null, '1700');

Insert into DEPARTMENTS

(DEPARTMENT\_ID, DEPARTMENT\_NAME, MANAGER\_ID, LOCATION\_ID)

Values

('100', 'Finance', null, '1700');

Insert into DEPARTMENTS

(DEPARTMENT\_ID, DEPARTMENT\_NAME, MANAGER\_ID, LOCATION\_ID)

Values

('110', 'Accounting', null, '1700');

COMMIT;

SET DEFINE OFF;

Insert into EMPLOYEES

(EMPLOYEE\_ID, FIRST\_NAME, LAST\_NAME, EMAIL, PHONE\_NUMBER,

HIRE\_DATE, JOB\_ID, SALARY, DEPARTMENT\_ID)

Values

(100, 'Steven', 'King', 'SKING', '515.123.4567',

TO\_DATE('06/17/1987 00:00:00', 'MM/DD/YYYY HH24:MI:SS'), 'AD\_PRES', 24000, 90);

Insert into EMPLOYEES

(EMPLOYEE\_ID, FIRST\_NAME, LAST\_NAME, EMAIL, PHONE\_NUMBER,

HIRE\_DATE, JOB\_ID, SALARY, MANAGER\_ID, DEPARTMENT\_ID)

Values

(103, 'Alexander', 'Hunold', 'AHUNOLD', '590.423.4567',

TO\_DATE('01/03/1990 00:00:00', 'MM/DD/YYYY HH24:MI:SS'), 'IT\_PROG', 9000, null, 60);

Insert into EMPLOYEES

(EMPLOYEE\_ID, FIRST\_NAME, LAST\_NAME, EMAIL, PHONE\_NUMBER,

HIRE\_DATE, JOB\_ID, SALARY, MANAGER\_ID, DEPARTMENT\_ID)

Values

(104, 'Bruce', 'Ernst', 'BERNST', '590.423.4568',

TO\_DATE('05/21/1991 00:00:00', 'MM/DD/YYYY HH24:MI:SS'), 'IT\_PROG', 6000, null, 60);

Insert into EMPLOYEES

(EMPLOYEE\_ID, FIRST\_NAME, LAST\_NAME, EMAIL, PHONE\_NUMBER,

HIRE\_DATE, JOB\_ID, SALARY, MANAGER\_ID, DEPARTMENT\_ID)

Values

(105, 'David', 'Austin', 'DAUSTIN', '590.423.4569',

TO\_DATE('06/25/1997 00:00:00', 'MM/DD/YYYY HH24:MI:SS'), 'IT\_PROG', 4800, null, 60);

Insert into EMPLOYEES

(EMPLOYEE\_ID, FIRST\_NAME, LAST\_NAME, EMAIL, PHONE\_NUMBER,

HIRE\_DATE, JOB\_ID, SALARY, MANAGER\_ID, DEPARTMENT\_ID)

Values

(106, 'Valli', 'Pataballa', 'VPATABAL', '590.423.4560',

TO\_DATE('02/05/1998 00:00:00', 'MM/DD/YYYY HH24:MI:SS'), 'IT\_PROG', 4800, null, 60);

Insert into EMPLOYEES

(EMPLOYEE\_ID, FIRST\_NAME, LAST\_NAME, EMAIL, PHONE\_NUMBER,

HIRE\_DATE, JOB\_ID, SALARY, MANAGER\_ID, DEPARTMENT\_ID)

Values

(107, 'Diana', 'Lorentz', 'DLORENTZ', '590.423.5567',

TO\_DATE('02/07/1999 00:00:00', 'MM/DD/YYYY HH24:MI:SS'), 'IT\_PROG', 4200, null, 60);

Insert into EMPLOYEES

(EMPLOYEE\_ID, FIRST\_NAME, LAST\_NAME, EMAIL, PHONE\_NUMBER,

HIRE\_DATE, JOB\_ID, SALARY, MANAGER\_ID, DEPARTMENT\_ID)

Values

(108, 'Nancy', 'Greenberg', 'NGREENBE', '515.124.4569',

TO\_DATE('08/17/1994 00:00:00', 'MM/DD/YYYY HH24:MI:SS'), 'FI\_MGR', 12000, null, 100);

Insert into EMPLOYEES

(EMPLOYEE\_ID, FIRST\_NAME, LAST\_NAME, EMAIL, PHONE\_NUMBER,

HIRE\_DATE, JOB\_ID, SALARY, MANAGER\_ID, DEPARTMENT\_ID)

Values

(109, 'Daniel', 'Faviet', 'DFAVIET', '515.124.4169',

TO\_DATE('08/16/1994 00:00:00', 'MM/DD/YYYY HH24:MI:SS'), 'FI\_ACCOUNT', 9000, null, 100);

Insert into EMPLOYEES

(EMPLOYEE\_ID, FIRST\_NAME, LAST\_NAME, EMAIL, PHONE\_NUMBER,

HIRE\_DATE, JOB\_ID, SALARY, MANAGER\_ID, DEPARTMENT\_ID)

Values

(110, 'John', 'Chen', 'JCHEN', '515.124.4269',

TO\_DATE('09/28/1997 00:00:00', 'MM/DD/YYYY HH24:MI:SS'), 'FI\_ACCOUNT', 8200, null, 100);

Insert into EMPLOYEES

(EMPLOYEE\_ID, FIRST\_NAME, LAST\_NAME, EMAIL, PHONE\_NUMBER,

HIRE\_DATE, JOB\_ID, SALARY, MANAGER\_ID, DEPARTMENT\_ID)

Values

(111, 'Ismael', 'Sciarra', 'ISCIARRA', '515.124.4369',

TO\_DATE('09/30/1997 00:00:00', 'MM/DD/YYYY HH24:MI:SS'), 'FI\_ACCOUNT', 7700, null, 100);

Insert into EMPLOYEES

(EMPLOYEE\_ID, FIRST\_NAME, LAST\_NAME, EMAIL, PHONE\_NUMBER,

HIRE\_DATE, JOB\_ID, SALARY, MANAGER\_ID, DEPARTMENT\_ID)

Values

(112, 'Jose Manuel', 'Urman', 'JMURMAN', '515.124.4469',

TO\_DATE('03/07/1998 00:00:00', 'MM/DD/YYYY HH24:MI:SS'), 'FI\_ACCOUNT', 7800, null, 100);

Insert into EMPLOYEES

(EMPLOYEE\_ID, FIRST\_NAME, LAST\_NAME, EMAIL, PHONE\_NUMBER,

HIRE\_DATE, JOB\_ID, SALARY, MANAGER\_ID, DEPARTMENT\_ID)

Values

(113, 'Luis', 'Popp', 'LPOPP', '515.124.4567',

TO\_DATE('12/07/1999 00:00:00', 'MM/DD/YYYY HH24:MI:SS'), 'FI\_ACCOUNT', 6900, null, 100);

Insert into EMPLOYEES

(EMPLOYEE\_ID, FIRST\_NAME, LAST\_NAME, EMAIL, PHONE\_NUMBER,

HIRE\_DATE, JOB\_ID, SALARY, MANAGER\_ID, DEPARTMENT\_ID)

Values

(114, 'Den', 'Raphaely', 'DRAPHEAL', '515.127.4561',

TO\_DATE('12/07/1994 00:00:00', 'MM/DD/YYYY HH24:MI:SS'), 'PU\_MAN', 11000, null, 30);

Insert into EMPLOYEES

(EMPLOYEE\_ID, FIRST\_NAME, LAST\_NAME, EMAIL, PHONE\_NUMBER,

HIRE\_DATE, JOB\_ID, SALARY, MANAGER\_ID, DEPARTMENT\_ID)

Values

(125, 'Julia', 'Nayer', 'JNAYER', '650.124.1214',

TO\_DATE('07/16/1997 00:00:00', 'MM/DD/YYYY HH24:MI:SS'), 'ST\_CLERK', 3200, null, 50);

Insert into EMPLOYEES

(EMPLOYEE\_ID, FIRST\_NAME, LAST\_NAME, EMAIL, PHONE\_NUMBER,

HIRE\_DATE, JOB\_ID, SALARY, MANAGER\_ID, DEPARTMENT\_ID)

Values

(126, 'Irene', 'Mikkilineni', 'IMIKKILI', '650.124.1224',

TO\_DATE('09/28/1998 00:00:00', 'MM/DD/YYYY HH24:MI:SS'), 'ST\_CLERK', 2700, null, 50);

Insert into EMPLOYEES

(EMPLOYEE\_ID, FIRST\_NAME, LAST\_NAME, EMAIL, PHONE\_NUMBER,

HIRE\_DATE, JOB\_ID, SALARY, MANAGER\_ID, DEPARTMENT\_ID)

Values

(127, 'James', 'Landry', 'JLANDRY', '650.124.1334',

TO\_DATE('01/14/1999 00:00:00', 'MM/DD/YYYY HH24:MI:SS'), 'ST\_CLERK', 2400, null, 50);

Insert into EMPLOYEES

(EMPLOYEE\_ID, FIRST\_NAME, LAST\_NAME, EMAIL, PHONE\_NUMBER,

HIRE\_DATE, JOB\_ID, SALARY, MANAGER\_ID, DEPARTMENT\_ID)

Values

(128, 'Steven', 'Markle', 'SMARKLE', '650.124.1434',

TO\_DATE('03/08/2000 00:00:00', 'MM/DD/YYYY HH24:MI:SS'), 'ST\_CLERK', 2200, null, 50);

Insert into EMPLOYEES

(EMPLOYEE\_ID, FIRST\_NAME, LAST\_NAME, EMAIL, PHONE\_NUMBER,

HIRE\_DATE, JOB\_ID, SALARY, MANAGER\_ID, DEPARTMENT\_ID)

Values

(129, 'Laura', 'Bissot', 'LBISSOT', '650.124.5234',

TO\_DATE('08/20/1997 00:00:00', 'MM/DD/YYYY HH24:MI:SS'), 'ST\_CLERK', 3300, null, 50);

Insert into EMPLOYEES

(EMPLOYEE\_ID, FIRST\_NAME, LAST\_NAME, EMAIL, PHONE\_NUMBER,

HIRE\_DATE, JOB\_ID, SALARY, MANAGER\_ID, DEPARTMENT\_ID)

Values

(130, 'Mozhe', 'Atkinson', 'MATKINSO', '650.124.6234',

TO\_DATE('10/30/1997 00:00:00', 'MM/DD/YYYY HH24:MI:SS'), 'ST\_CLERK', 2800, null, 50);

Insert into EMPLOYEES

(EMPLOYEE\_ID, FIRST\_NAME, LAST\_NAME, EMAIL, PHONE\_NUMBER,

HIRE\_DATE, JOB\_ID, SALARY, MANAGER\_ID, DEPARTMENT\_ID)

Values

(131, 'James', 'Marlow', 'JAMRLOW', '650.124.7234',

TO\_DATE('02/16/1997 00:00:00', 'MM/DD/YYYY HH24:MI:SS'), 'ST\_CLERK', 2500, null, 50);

Insert into EMPLOYEES

(EMPLOYEE\_ID, FIRST\_NAME, LAST\_NAME, EMAIL, PHONE\_NUMBER,

HIRE\_DATE, JOB\_ID, SALARY, MANAGER\_ID, DEPARTMENT\_ID)

Values

(132, 'TJ', 'Olson', 'TJOLSON', '650.124.8234',

TO\_DATE('04/10/1999 00:00:00', 'MM/DD/YYYY HH24:MI:SS'), 'ST\_CLERK', 2100, null, 50);

Insert into EMPLOYEES

(EMPLOYEE\_ID, FIRST\_NAME, LAST\_NAME, EMAIL, PHONE\_NUMBER,

HIRE\_DATE, JOB\_ID, SALARY, MANAGER\_ID, DEPARTMENT\_ID)

Values

(133, 'Jason', 'Mallin', 'JMALLIN', '650.127.1934',

TO\_DATE('06/14/1996 00:00:00', 'MM/DD/YYYY HH24:MI:SS'), 'ST\_CLERK', 3300, null, 50);

Insert into EMPLOYEES

(EMPLOYEE\_ID, FIRST\_NAME, LAST\_NAME, EMAIL, PHONE\_NUMBER,

HIRE\_DATE, JOB\_ID, SALARY, MANAGER\_ID, DEPARTMENT\_ID)

Values

(134, 'Michael', 'Rogers', 'MROGERS', '650.127.1834',

TO\_DATE('08/26/1998 00:00:00', 'MM/DD/YYYY HH24:MI:SS'), 'ST\_CLERK', 2900, null, 50);

Insert into EMPLOYEES

(EMPLOYEE\_ID, FIRST\_NAME, LAST\_NAME, EMAIL, PHONE\_NUMBER,

HIRE\_DATE, JOB\_ID, SALARY, MANAGER\_ID, DEPARTMENT\_ID)

Values

(135, 'Ki', 'Gee', 'KGEE', '650.127.1734',

TO\_DATE('12/12/1999 00:00:00', 'MM/DD/YYYY HH24:MI:SS'), 'ST\_CLERK', 2400, null, 50);

Insert into EMPLOYEES

(EMPLOYEE\_ID, FIRST\_NAME, LAST\_NAME, EMAIL, PHONE\_NUMBER,

HIRE\_DATE, JOB\_ID, SALARY, MANAGER\_ID, DEPARTMENT\_ID)

Values

(136, 'Hazel', 'Philtanker', 'HPHILTAN', '650.127.1634',

TO\_DATE('02/06/2000 00:00:00', 'MM/DD/YYYY HH24:MI:SS'), 'ST\_CLERK', 2200, null, 50);

Insert into EMPLOYEES

(EMPLOYEE\_ID, FIRST\_NAME, LAST\_NAME, EMAIL, PHONE\_NUMBER,

HIRE\_DATE, JOB\_ID, SALARY, MANAGER\_ID, DEPARTMENT\_ID)

Values

(137, 'Renske', 'Ladwig', 'RLADWIG', '650.121.1234',

TO\_DATE('07/14/1995 00:00:00', 'MM/DD/YYYY HH24:MI:SS'), 'ST\_CLERK', 3600, null, 50);

Insert into EMPLOYEES

(EMPLOYEE\_ID, FIRST\_NAME, LAST\_NAME, EMAIL, PHONE\_NUMBER,

HIRE\_DATE, JOB\_ID, SALARY, MANAGER\_ID, DEPARTMENT\_ID)

Values

(138, 'Stephen', 'Stiles', 'SSTILES', '650.121.2034',

TO\_DATE('10/26/1997 00:00:00', 'MM/DD/YYYY HH24:MI:SS'), 'ST\_CLERK', 3200, null, 50);

Insert into EMPLOYEES

(EMPLOYEE\_ID, FIRST\_NAME, LAST\_NAME, EMAIL, PHONE\_NUMBER,

HIRE\_DATE, JOB\_ID, SALARY, MANAGER\_ID, DEPARTMENT\_ID)

Values

(139, 'John', 'Seo', 'JSEO', '650.121.2019',

TO\_DATE('02/12/1998 00:00:00', 'MM/DD/YYYY HH24:MI:SS'), 'ST\_CLERK', 2700, null, 50);

Insert into EMPLOYEES

(EMPLOYEE\_ID, FIRST\_NAME, LAST\_NAME, EMAIL, PHONE\_NUMBER,

HIRE\_DATE, JOB\_ID, SALARY, MANAGER\_ID, DEPARTMENT\_ID)

Values

(140, 'Joshua', 'Patel', 'JPATEL', '650.121.1834',

TO\_DATE('04/06/1998 00:00:00', 'MM/DD/YYYY HH24:MI:SS'), 'ST\_CLERK', 2500, null, 50);

Insert into EMPLOYEES

(EMPLOYEE\_ID, FIRST\_NAME, LAST\_NAME, EMAIL, PHONE\_NUMBER,

HIRE\_DATE, JOB\_ID, SALARY, MANAGER\_ID, DEPARTMENT\_ID)

Values

(141, 'Trenna', 'Rajs', 'TRAJS', '650.121.8009',

TO\_DATE('10/17/1995 00:00:00', 'MM/DD/YYYY HH24:MI:SS'), 'ST\_CLERK', 3500, null, 50);

Insert into EMPLOYEES

(EMPLOYEE\_ID, FIRST\_NAME, LAST\_NAME, EMAIL, PHONE\_NUMBER,

HIRE\_DATE, JOB\_ID, SALARY, MANAGER\_ID, DEPARTMENT\_ID)

Values

(142, 'Curtis', 'Davies', 'CDAVIES', '650.121.2994',

TO\_DATE('01/29/1997 00:00:00', 'MM/DD/YYYY HH24:MI:SS'), 'ST\_CLERK', 3100, null, 50);

Insert into EMPLOYEES

(EMPLOYEE\_ID, FIRST\_NAME, LAST\_NAME, EMAIL, PHONE\_NUMBER,

HIRE\_DATE, JOB\_ID, SALARY, MANAGER\_ID, DEPARTMENT\_ID)

Values

(143, 'Randall', 'Matos', 'RMATOS', '650.121.2874',

TO\_DATE('03/15/1998 00:00:00', 'MM/DD/YYYY HH24:MI:SS'), 'ST\_CLERK', 2600, null, 50);

Insert into EMPLOYEES

(EMPLOYEE\_ID, FIRST\_NAME, LAST\_NAME, EMAIL, PHONE\_NUMBER,

HIRE\_DATE, JOB\_ID, SALARY, MANAGER\_ID, DEPARTMENT\_ID)

Values

(144, 'Peter', 'Vargas', 'PVARGAS', '650.121.2004',

TO\_DATE('07/09/1998 00:00:00', 'MM/DD/YYYY HH24:MI:SS'), 'ST\_CLERK', 2500, null, 50);

Insert into EMPLOYEES

(EMPLOYEE\_ID, FIRST\_NAME, LAST\_NAME, EMAIL, PHONE\_NUMBER,

HIRE\_DATE, JOB\_ID, SALARY, MANAGER\_ID, DEPARTMENT\_ID)

Values

(202, 'Pat', 'Fay', 'PFAY', '603.123.6666',

TO\_DATE('08/17/1997 00:00:00', 'MM/DD/YYYY HH24:MI:SS'), 'MK\_REP', 6000, null, 20);

Insert into EMPLOYEES

(EMPLOYEE\_ID, FIRST\_NAME, LAST\_NAME, EMAIL, PHONE\_NUMBER,

HIRE\_DATE, JOB\_ID, SALARY, MANAGER\_ID, DEPARTMENT\_ID)

Values

(204, 'Hermann', 'Baer', 'HBAER', '515.123.8888',

TO\_DATE('06/07/1994 00:00:00', 'MM/DD/YYYY HH24:MI:SS'), 'PR\_REP', 10000, null, 70);

Insert into EMPLOYEES

(EMPLOYEE\_ID, FIRST\_NAME, LAST\_NAME, EMAIL, PHONE\_NUMBER,

HIRE\_DATE, JOB\_ID, SALARY, MANAGER\_ID, DEPARTMENT\_ID)

Values

(101, 'Neena', 'Kochhar', 'NKOCHHAR', '515.123.4568',

TO\_DATE('09/21/1989 00:00:00', 'MM/DD/YYYY HH24:MI:SS'), 'AD\_VP', 17000, null, 90);

Insert into EMPLOYEES

(EMPLOYEE\_ID, FIRST\_NAME, LAST\_NAME, EMAIL, PHONE\_NUMBER,

HIRE\_DATE, JOB\_ID, SALARY, MANAGER\_ID, DEPARTMENT\_ID)

Values

(102, 'Lex', 'De Haan', 'LDEHAAN', '515.123.4569',

TO\_DATE('01/13/1993 00:00:00', 'MM/DD/YYYY HH24:MI:SS'), 'AD\_VP', 17000, null, 90);

Insert into EMPLOYEES

(EMPLOYEE\_ID, FIRST\_NAME, LAST\_NAME, EMAIL, PHONE\_NUMBER,

HIRE\_DATE, JOB\_ID, SALARY, MANAGER\_ID, DEPARTMENT\_ID)

Values

(120, 'Matthew', 'Weiss', 'MWEISS', '650.123.1234',

TO\_DATE('07/18/1996 00:00:00', 'MM/DD/YYYY HH24:MI:SS'), 'ST\_MAN', 8000, null, 50);

Insert into EMPLOYEES

(EMPLOYEE\_ID, FIRST\_NAME, LAST\_NAME, EMAIL, PHONE\_NUMBER,

HIRE\_DATE, JOB\_ID, SALARY, MANAGER\_ID, DEPARTMENT\_ID)

Values

(121, 'Adam', 'Fripp', 'AFRIPP', '650.123.2234',

TO\_DATE('04/10/1997 00:00:00', 'MM/DD/YYYY HH24:MI:SS'), 'ST\_MAN', 8200, null, 50);

Insert into EMPLOYEES

(EMPLOYEE\_ID, FIRST\_NAME, LAST\_NAME, EMAIL, PHONE\_NUMBER,

HIRE\_DATE, JOB\_ID, SALARY, MANAGER\_ID, DEPARTMENT\_ID)

Values

(122, 'Payam', 'Kaufling', 'PKAUFLIN', '650.123.3234',

TO\_DATE('05/01/1995 00:00:00', 'MM/DD/YYYY HH24:MI:SS'), 'ST\_MAN', 7900, null, 50);

Insert into EMPLOYEES

(EMPLOYEE\_ID, FIRST\_NAME, LAST\_NAME, EMAIL, PHONE\_NUMBER,

HIRE\_DATE, JOB\_ID, SALARY, MANAGER\_ID, DEPARTMENT\_ID)

Values

(123, 'Shanta', 'Vollman', 'SVOLLMAN', '650.123.4234',

TO\_DATE('10/10/1997 00:00:00', 'MM/DD/YYYY HH24:MI:SS'), 'ST\_MAN', 6500, null, 50);

Insert into EMPLOYEES

(EMPLOYEE\_ID, FIRST\_NAME, LAST\_NAME, EMAIL, PHONE\_NUMBER,

HIRE\_DATE, JOB\_ID, SALARY, MANAGER\_ID, DEPARTMENT\_ID)

Values

(124, 'Kevin', 'Mourgos', 'KMOURGOS', '650.123.5234',

TO\_DATE('11/16/1999 00:00:00', 'MM/DD/YYYY HH24:MI:SS'), 'ST\_MAN', 5800, null, 50);

Insert into EMPLOYEES

(EMPLOYEE\_ID, FIRST\_NAME, LAST\_NAME, EMAIL, PHONE\_NUMBER,

HIRE\_DATE, JOB\_ID, SALARY, MANAGER\_ID, DEPARTMENT\_ID)

Values

(201, 'Michael', 'Hartstein', 'MHARTSTE', '515.123.5555',

TO\_DATE('02/17/1996 00:00:00', 'MM/DD/YYYY HH24:MI:SS'), 'MK\_MAN', 13000, null, 20);

Insert into EMPLOYEES

(EMPLOYEE\_ID, FIRST\_NAME, LAST\_NAME, EMAIL, PHONE\_NUMBER,

HIRE\_DATE, JOB\_ID, SALARY, MANAGER\_ID, DEPARTMENT\_ID)

Values

(205, 'Shelley', 'Higgins', 'SHIGGINS', '515.123.8080',

TO\_DATE('06/07/1994 00:00:00', 'MM/DD/YYYY HH24:MI:SS'), 'AC\_MGR', 12000, null, 110);

Insert into EMPLOYEES

(EMPLOYEE\_ID, FIRST\_NAME, LAST\_NAME, EMAIL, PHONE\_NUMBER,

HIRE\_DATE, JOB\_ID, SALARY, MANAGER\_ID, DEPARTMENT\_ID)

Values

(200, 'Jennifer', 'Whalen', 'JWHALEN', '515.123.4444',

TO\_DATE('09/17/1987 00:00:00', 'MM/DD/YYYY HH24:MI:SS'), 'AD\_ASST', 4400, null, 10);

COMMIT;

SET DEFINE OFF;

Insert into JOB\_HISTORY

(EMPLOYEE\_ID, START\_DATE, END\_DATE, JOB\_ID, DEPARTMENT\_ID)

Values

(100, TO\_DATE('06/17/1987 00:00:00', 'MM/DD/YYYY HH24:MI:SS'), TO\_DATE('07/24/1998 00:00:00', 'MM/DD/YYYY HH24:MI:SS'), 'IT\_PROG', 90);

COMMIT;

SET DEFINE OFF;

UPDATE EMPLOYEES SET MANAGER\_ID = 100 WHERE EMPLOYEE\_ID IN (101,102,114,120,121,122,123,124,201);

UPDATE EMPLOYEES SET MANAGER\_ID = 102 WHERE EMPLOYEE\_ID = 103;

UPDATE EMPLOYEES SET MANAGER\_ID = 103 WHERE EMPLOYEE\_ID IN (104,105,106,107);

UPDATE EMPLOYEES SET MANAGER\_ID = 101 WHERE EMPLOYEE\_ID = 108;

UPDATE EMPLOYEES SET MANAGER\_ID = 108 WHERE EMPLOYEE\_ID IN (109,110,111,112,113);

UPDATE EMPLOYEES SET MANAGER\_ID = 120 WHERE EMPLOYEE\_ID IN (125,126,127,128);

UPDATE EMPLOYEES SET MANAGER\_ID = 121 WHERE EMPLOYEE\_ID IN (129,130,131,132);

UPDATE EMPLOYEES SET MANAGER\_ID = 122 WHERE EMPLOYEE\_ID IN (133,134,135,136);

UPDATE EMPLOYEES SET MANAGER\_ID = 123 WHERE EMPLOYEE\_ID IN (137,138,139,140);

UPDATE EMPLOYEES SET MANAGER\_ID = 124 WHERE EMPLOYEE\_ID IN (141,142,143);

COMMIT;

SET DEFINE OFF;

UPDATE DEPARTMENTS SET MANAGER\_ID = 200 WHERE DEPARTMENT\_ID = 10;

UPDATE DEPARTMENTS SET MANAGER\_ID = 201 WHERE DEPARTMENT\_ID = 20;

UPDATE DEPARTMENTS SET MANAGER\_ID = 114 WHERE DEPARTMENT\_ID = 30;

UPDATE DEPARTMENTS SET MANAGER\_ID = 121 WHERE DEPARTMENT\_ID = 50;

UPDATE DEPARTMENTS SET MANAGER\_ID = 204 WHERE DEPARTMENT\_ID = 70;

UPDATE DEPARTMENTS SET MANAGER\_ID = 100 WHERE DEPARTMENT\_ID = 90;

UPDATE DEPARTMENTS SET MANAGER\_ID = 108 WHERE DEPARTMENT\_ID = 100;

UPDATE DEPARTMENTS SET MANAGER\_ID = 205 WHERE DEPARTMENT\_ID = 110;

COMMIT;

As the development environment is setup, you shall now start exploring the solutions for the different requirements.

**Requirement 1: HR Manager wants to update the salary of employees**

Based on the latest compensation review, the company has decided to give an annual hike of 10% to all its employees and an additional hike of 5% to its employees who have joined before '01-Jan-95'.   
HR manager wants to calculate the new salary of Alexander based on these rules, update the database and also display a report in the below format.

Name : <First\_Name> <Last\_Name>  
Date of Joining : <Hire\_Date>  
Current Salary : <Salary>  
Incremented Salary : <Calculated Salary>

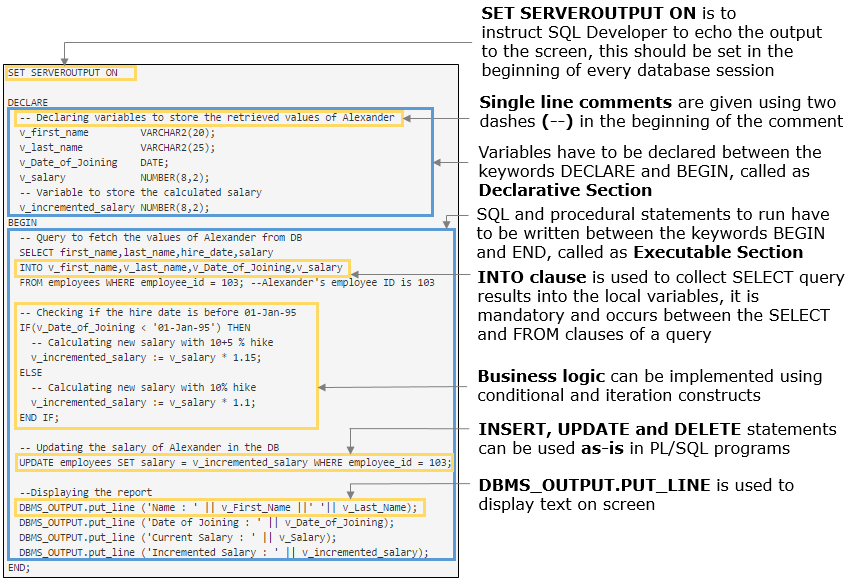
**Execute the below SQL query and observe the table data in Query Result tab:**

1. SELECT \* FROM employees;

**Execute the PL/SQL solution in SQL Developer and observe the result in Script Output tab:**

1. SET SERVEROUTPUT ON
3. DECLARE
4. -- Declaring variables to store the retrieved values of Alexander
5. v\_first\_name VARCHAR2(20);
6. v\_last\_name VARCHAR2(25);
7. v\_Date\_of\_Joining DATE;
8. v\_salary NUMBER(8,2);
9. -- Variable to store the calculated salary
10. v\_incremented\_salary NUMBER(8,2);
11. BEGIN
12. -- Query to fetch the values of Alexander from DB
13. SELECT first\_name,last\_name,hire\_date,salary
14. INTO v\_first\_name,v\_last\_name,v\_Date\_of\_Joining,v\_salary
15. FROM employees WHERE employee\_id = 103; --Alexander's employee ID is 103
17. -- Checking if the hire date is before 01-Jan-95
18. IF(v\_Date\_of\_Joining < '01-Jan-95') THEN
19. -- Calculating new salary with 10+5 % hike
20. v\_incremented\_salary := v\_salary \* 1.15;
21. ELSE
22. -- Calculating new salary with 10% hike
23. v\_incremented\_salary := v\_salary \* 1.1;
24. END IF;
26. -- Updating the salary of Alexander in the DB
27. UPDATE employees SET salary = v\_incremented\_salary WHERE employee\_id = 103;
29. --Displaying the report
30. DBMS\_OUTPUT.put\_line ('Name : ' || v\_First\_Name ||' '|| v\_Last\_Name);
31. DBMS\_OUTPUT.put\_line ('Date of Joining : ' || v\_Date\_of\_Joining);
32. DBMS\_OUTPUT.put\_line ('Current Salary : ' || v\_Salary);
33. DBMS\_OUTPUT.put\_line ('Incremented Salary : ' || v\_incremented\_salary);
34. END;

**Requirement 1 : HR Manager wants to update the salary of employees: Code anatomy**



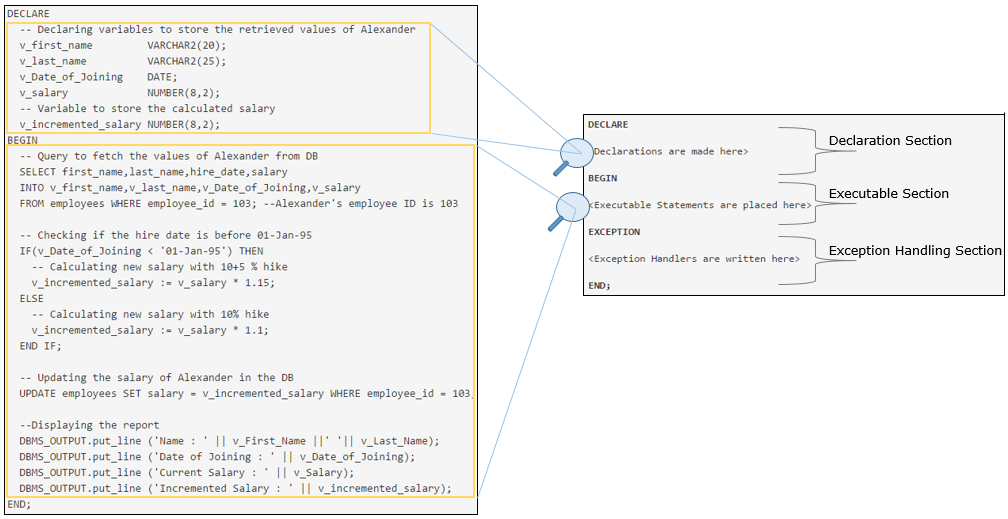
The code you just executed is the standard PL/SQL program unit called as block.

A PL/SQL block is defined by the keywords DECLARE, BEGIN, EXCEPTION and END which break up the block into three sections:

* **Declarative:** has statements that declare variables, constants, and other code elements, which can then be used within that block
* **Executable:** has statements that are run when the block is executed
* **Exception handling:** has statements that can handle any runtime exceptions from the executable section

**Note :**Only the executable section of a PL/SQL block is mandatory, declarative and exception handling sections can be used if required.

**Structure of a PL/SQL block:**



## Problem Statement:

# Requirement 1: HR Manager wants to update the salary of employees

**Revisiting requirement 1:**Test the existing program to check if salary is properly calculated for employees who have joined after '01-Jan-95'.

Execute the below query and observe the hire date and salary of Diana Lorentz, employee ID 107.

1. SELECT first\_name,last\_name,hire\_date,salary FROM employees WHERE employee\_id = 107;

Modify the PL/SQL block to calculate and display the salary of Diana Lorentz, employee ID 107.

1. DECLARE
2. -- Declaring variables to store the retrieved values of Alexander
3. v\_first\_name VARCHAR2(20);
4. v\_last\_name VARCHAR2(25);
5. v\_Date\_of\_Joining DATE;
6. v\_salary NUMBER(8,2);
7. -- Variable to store the calculated salary
8. v\_incremented\_salary NUMBER(8,2);
9. BEGIN
10. -- Query to fetch the values of Alexander from DB
11. SELECT first\_name,last\_name,hire\_date,salary
12. INTO v\_first\_name,v\_last\_name,v\_Date\_of\_Joining,v\_salary
13. FROM employees WHERE employee\_id = 103; --Alexander's employee ID is 103
15. -- Checking if the hire date is before 01-Jan-95
16. IF(v\_Date\_of\_Joining < '01-Jan-95') THEN
17. -- Calculating new salary with 10+5 % hike
18. v\_incremented\_salary := v\_salary \* 1.15;
19. ELSE
20. -- Calculating new salary with 10% hike
21. v\_incremented\_salary := v\_salary \* 1.1;
22. END IF;
24. -- Updating the salary of Alexander in the DB
25. UPDATE employees SET salary = v\_incremented\_salary WHERE employee\_id = 103;
27. --Displaying the report
28. DBMS\_OUTPUT.put\_line ('Name : ' || v\_First\_Name ||' '|| v\_Last\_Name);
29. DBMS\_OUTPUT.put\_line ('Date of Joining : ' || v\_Date\_of\_Joining);
30. DBMS\_OUTPUT.put\_line ('Current Salary : ' || v\_Salary);
31. DBMS\_OUTPUT.put\_line ('Incremented Salary : ' || v\_incremented\_salary);
32. END;

Execute the below query and verify the salary of Diana Lorentz, employee ID 107.

1. SELECT first\_name,last\_name,hire\_date,salary FROM employees WHERE employee\_id = 107;

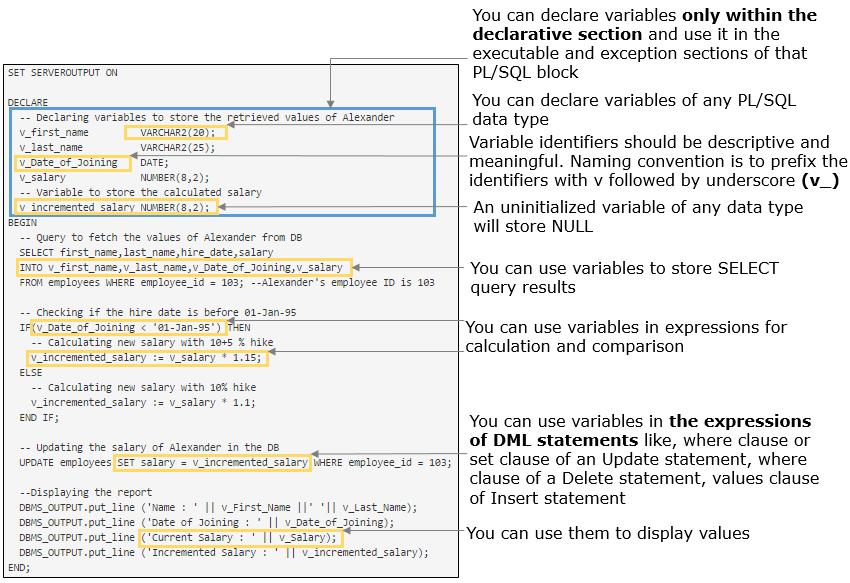
Unlike SQL, PL/SQL supports variables. You can declare a variable only in the declarative section and use it anywhere within the PL/SQL block to store the SQL retrieved values and/or in expressions for calculations.

**Syntax for variable declaration:**

1. variable\_name datatype [ optional assignment ];

**Variables used in the program for Requirement 1:**

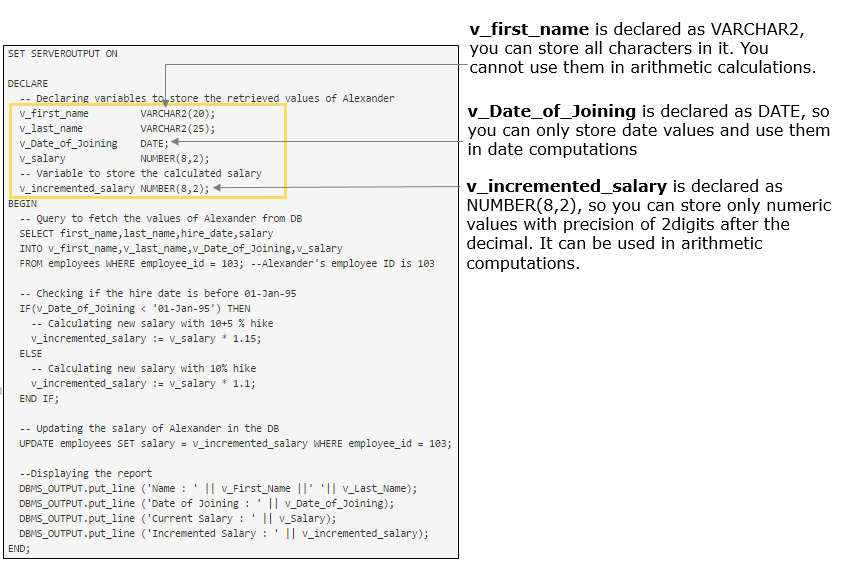
Variables v\_first\_name, v\_last\_name, v\_Date\_of\_Joining and v\_salary are used to store and display the retrieved values from the database. v\_incremented\_salary, v\_Date\_of\_Joining and v\_salary are used in computations and comparisons.



Variables are used to store values but can you store any type of value in any variable or can any variable be used for any type of computation?

No, datatype of the variables determine the storage format and memory allocation for a variable, range of valid values that can be stored in a variable and operations that can be performed with a variable.

**Variables of different datatypes used in the program for Requirement 1:**



Highlights:

* Requirement 2: An employee wants to view the first name and phone number of other employees
* Declare variables to store these values and display.

Demosteps:

**Step 1**: Execute the below code in SQL Developer and observe the output.

1. --to instruct SQL Developer to echo the output to the screen
2. SET SERVEROUTPUT ON;
4. -- PL/SQL block for the requirement
5. DECLARE
6. v\_employee\_id NUMBER(6) := 100;
7. v\_first\_name VARCHAR2(20);
8. v\_phone\_number VARCHAR2(25);
9. BEGIN
10. */\*Details of employee with ID 100 will be retrieved and stored*
11. *in the variables using select statement\*/*
12. SELECT first\_name,phone\_number
13. INTO v\_first\_name,v\_phone\_number
14. FROM employees
15. WHERE employee\_id= v\_employee\_id;
16. --You will see how to write SQL statements in PL/SQL block in the next section
17. DBMS\_OUTPUT.PUT\_LINE ('Employee ID: ' || v\_employee\_id);
18. DBMS\_OUTPUT.PUT\_LINE ('First Name: ' || v\_first\_name);
19. DBMS\_OUTPUT.PUT\_LINE ('Contact Number: ' || v\_phone\_number);
20. END;

**Note:**

In PL/SQL, **:=** is used for assignment and **=** is used for comparison.

You can use **:=** or **DEFAULT** to initialize a variable in the **DECLARE** section.

**Multi line comments** are given by enclosing the lines between **/\***and **\*/**

Similar to constraints on columns of a table, you can add the below constraints to a PL/SQL variable to modify their behavior.

# Syntax of variable declaration with constraints

1. variable\_name [CONSTANT] datatype [NOT NULL] [:= | DEFAULT initial\_value]

**NOT NULL**prevents assigning a null value to the variable.

1. DECLARE
2. v\_salary   Number (10) **NOT NULL** := 10000;
3. BEGIN
4. DBMS\_OUTPUT.put\_line (v\_salary);
5. v\_salary:=NULL; **--This line will give compilation error saying expression is of wrong type**
6. END;

**CONSTANT**constraint prevents their value being altered in program.

1. DECLARE
2. v\_basic\_salary   CONSTANT VARCHAR2 (10) := 5000;
3. BEGIN
4. DBMS\_OUTPUT.put\_line (v\_basic\_salary);
5. v\_basic\_salary:=2000;
6. */\*above line will give compilation error expression 'V\_BASIC\_SALARY' cannot be used as an*
7. *assignment target \*/*
8. END;

**NOTE:**Variable must be initialized if declared with CONSTANT or NOT NULL constraint.

PL/SQL datatypes are classified into scalar and composite datatypes.

**Scalar Types** are used to store single values with no internal components.

In addition to supporting all the SQL datatypes, PL/SQL provides the below exclusive datatypes.

* BOOLEAN
* PLS\_INTEGER
* BINARY\_INTEGER
* REF CURSOR
* User-defined subtypes

**Composite Types** are structured datatypes that have components you can address independently. Records and collections are composite datatypes which will be covered later in the course.

**NOTE:** The basic datatypes and their required information are covered in this course. For other datatypes and more detailed information you can refer [Oracle Docs](https://docs.oracle.com/)

# Numeric Datatypes

* **PLS\_INTEGER** is used to store signed integers
* **NUMBER(p,s)** is mostly used for storing floating-point values but can even accommodate integer

# Character Datatypes

* **CHAR(n)** is a fixed length character datatype. The precision is specified in integer which represents the number of bytes
* **VARCHAR2(n)** is a variable length character datatype

# Date and Time

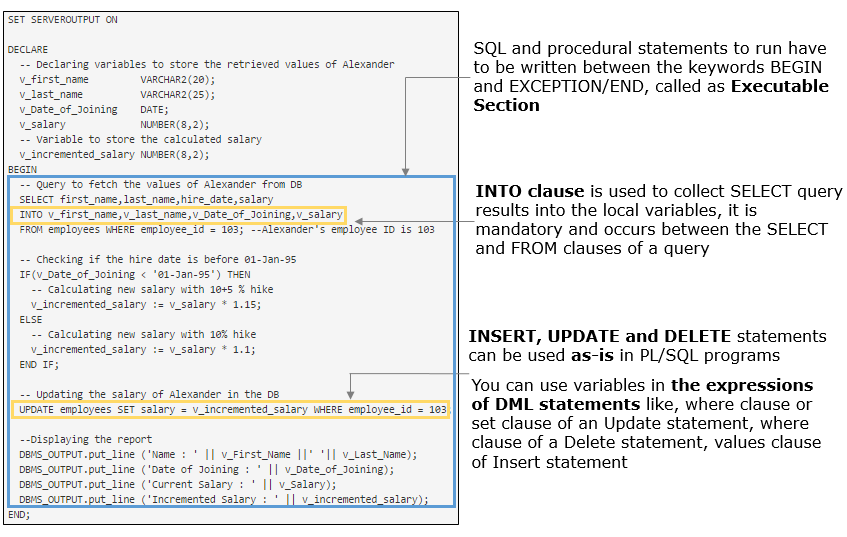
* **DATE**is used for storing date and time. It stores the century, year, month, day, hour, minute, second. Fractional seconds are not available in this datatype
* **TIMESTAMP**stores the date and time details much like DATE datatype, and in addition, it also provides the sub-second details up to nine digits. The default precision is six digits of subseconds

# Boolean

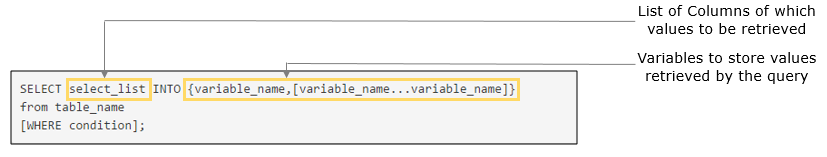
* **BOOLEAN**is used for storing logical values such as TRUE, FALSE or NULL. It is mainly used to store the comparison results. Do not attempt to print or display the value stored in Boolean variable, as it is not possible.When assigning values to Boolean variables, do not enclose in quotes

**Usage of SQL statements in the program for Requirement 1:**

Select statement is used to retrieve the values of Alexander, employee ID 103, and store it in the variables. Update statement is used to modify his salary in the database.



**Syntax for select statement inside PL/SQL with INTO clause:**



**INTO Clause**

INTO clause is used in PL/SQL blocks to collect SELECT query results into a local variable.The INTO clause is mandatory and occurs between the SELECT and FROM clauses.

There must be a variable corresponding to all the columns retrieved by SELECT statement and their corresponding datatype must match.

The form of the SELECT statement discussed here will result in an error if SELECT returns more than one record. Later in this course we will see how to retrieve multiple rows from SELECT statement.

 Revisiting requirement 2: An employee wants to view the first name and phone number of other employees

 Develop a PL/SQL program to retrieve and display the first name and phone number of employee with employee ID 100.

Demosteps:

**Step 1:** Execute the below code in SQL Developer and observe the output.

1. DECLARE
2. v\_employee\_id NUMBER (6):= 100;
3. v\_first\_name VARCHAR2(20);
4. v\_phone\_number VARCHAR2(25);
5. BEGIN
7. --Values of employee with id 100 are retrieved and stored in the variables
8. SELECT first\_name, phone\_number
9. INTO v\_first\_name, v\_phone\_number
10. FROM employees where employee\_id= v\_employee\_id;
12. DBMS\_OUTPUT.PUT\_LINE ('Employee ID: ' || v\_employee\_id);
13. DBMS\_OUTPUT.PUT\_LINE ('First Name: ' || v\_first\_name);
14. DBMS\_OUTPUT.PUT\_LINE ('Contact Number: ' || v\_phone\_number);
15. END;

## ProblemStatement:

# ****Revisiting requirement 2:**** An employee wants to view the first name and phone number of other employees

**Part 1:** Use the same program to retrieve and display the first name and phone number of employee with employee ID 21. Execute and observe the output.

Expected result: You will get an exception **ORA-01403: no data found**.

**Part 2:** In the same program, remove the WHERE clause of SELECT statement and then execute the program and observe the output.

Expected result: You will get an exception **ORA-01422: exact fetch returns more than requested number of rows**.

**Reason:** SELECT statement with INTO clause will result in an exception (runtime error), if the SELECT query returns zero or more than one record. You will learn how to handle such exceptions later.

**Note:** Later in this course you will also see how to retrieve multiple rows from SELECT statement and manipulate them.

## ProblemStatement:

# Requirement 2: An employee wants to view the first name and phone number of other employees

**Revisiting requirement 2:** The company has decided that an employee can also view the email ID and department ID of all other employees, in addition to first name and phone number.

Enhance the code to achieve the requirement.

1. DECLARE
2. v\_employee\_id NUMBER (6):= 104;
4. */\* Declare three variables with appropriate datatype*
5. *to store email id and department id\*/*
7. BEGIN
9. --Retrieve the values of employee ID 104 from the database using Select statement
11. DBMS\_OUTPUT.PUT\_LINE ('Employee ID: ' || v\_employee\_id);
12. -- Display the other details in similar format
14. END

## ProblemStatement:

# Requirement 3: Manager wants to view the joining date of other employees

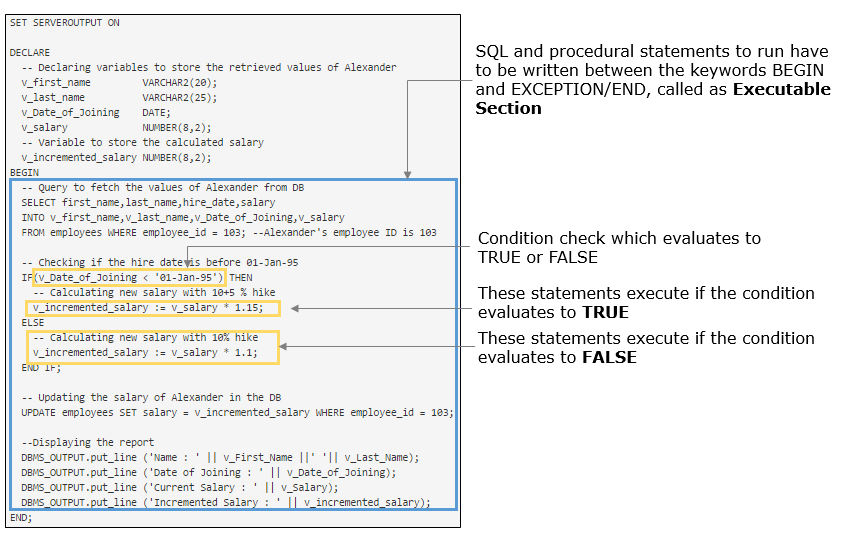
Manager wants to view the joining date, in addition to other details, of other employees. Retrieve and display the details of employee with ID 104.

Enhance the code developed in the previous requirement.

**Continuing with Requirement 1**, we had to decide the increment percentage based on the date of joining of the employee.

In such requirements, where you need to either run or skip a sequence of one or more statements, depending on a condition, you should go for conditional constructs. You test a Boolean condition and depending on if it proves to be TRUE or FALSE or NULL, you carry out some actions.

**Usage of conditional constructs in the program for Requirement 1:**



The PL/SQL IF statement has these three forms. The kind of if statement you use depends, of course, on what you want to do. Sometimes you will want to carry out an action if a condition is true, and do nothing otherwise; other times you may want to carry out some actions if a condition is true, and other actions if it’s false.

| **IF-THEN** | **IF-THEN-ELSE** | **IF-THEN-ELSIF** |
| --- | --- | --- |
| 1. BEGIN 2. IF <condition> THEN 3. STATEMENT 1; 4. . 5. . 6. STATEMENT N; 7. END IF; 8. END; | 1. BEGIN 2. IF <condition> THEN 3. STATEMENT 1; 4. . 5. . 6. STATEMENT N; 7. ELSE 8. STATEMENT 1; 9. . 10. . 11. STATEMENT N; 12. END IF; 13. END; | 1. BEGIN 2. IF <condition> THEN 3. STATEMENT 1; 4. . 5. . 6. STATEMENT N; 7. ELSIF <condition2> THEN 8. STATEMENT 1; 9. . 10. . 11. STATEMENT N; 12. ELSE 13. STATEMENT 1; 14. . 15. . 16. STATEMENT N; 17. END IF; 18. END; |

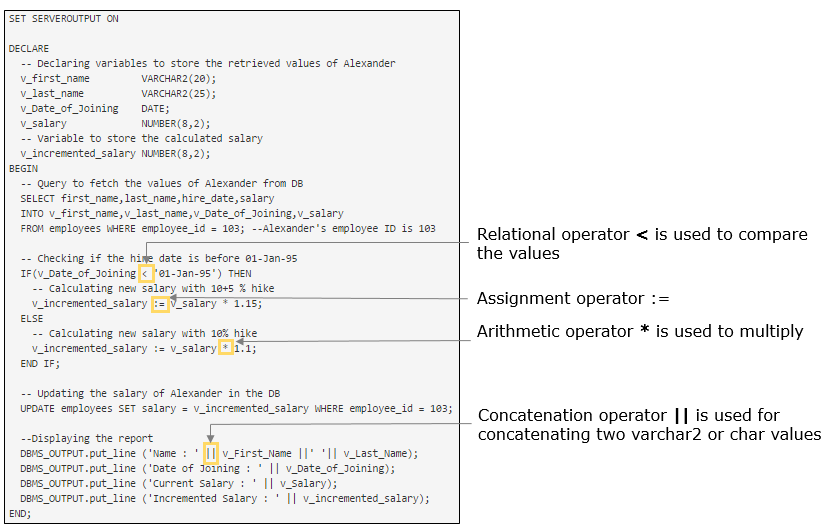
You have seen how to declare, store and display values in a variable, but to serve the purpose of data manipulation and calculation on data stored in variables we need operators.

Operators supported in PL/SQL are:

* Concatenation Operator ( || )
* Assignment operator (:=)
* Arithmetic Operators ( +, -, \*, /,\*\*)
* Relational Operators ( =, !=, <, >, <=, >=)
* Logical Operators (AND, OR and NOT)

You can refer [Oracle Docs](https://docs.oracle.com/) for more operators supported by PL/SQL

**Usage of operators in the program for Requirement 1:**



Highlights:

 Requirement 4: An employee wants to check if he/she is eligible to pay tax. Employees having salary greater than 8000 are eligible to pay tax

 Develop a PL/SQL program to check if Steven King, employee ID 100, is eligible to pay tax.

Demosteps:

**Step 1**: Execute the below code in SQL Developer and observe the output.

2. DECLARE
3. --variable to store the retrieved salary
4. v\_salary NUMBER(8,2);
5. v\_eligible\_salary CONSTANT NUMBER(6) := 8000;
6. v\_employee\_id NUMBER(3):=100;
7. BEGIN
8. --salary of Steven, employee id 100 is retrieved
9. SELECT salary INTO v\_salary FROM employees WHERE employee\_id=v\_employee\_id;
10. --retrieved value is checked as per the rule
11. IF v\_salary >v\_eligible\_salary THEN
12. DBMS\_OUTPUT.put\_line('Eligible to pay tax');
13. END IF;
14. END;

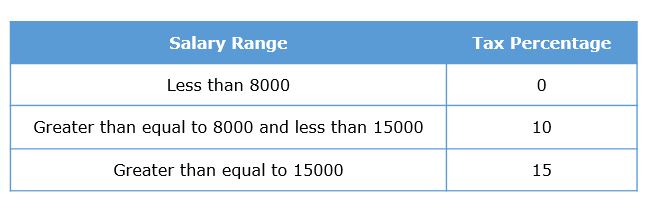
Highlights:

 Requirement 5: An employee wants to calculate his/her tax amount. Employees need to pay tax to the government based on the given salary slabs

 Develop a PL/SQL program to calculate and display the tax amount of Steven King, employee ID 100.

Demosteps:

Tax is computed based on the below rules:



**Step 1**: Execute the below code in SQL Developer and observe the output.

1. DECLARE
2. --used for tax calculation
3. p\_employee\_id  NUMBER(6):=100;
4. v\_salary NUMBER(8,2) ;
5. v\_tax\_percentage NUMBER(2);
6. v\_tax\_amount NUMBER(8,2);
7. BEGIN
8. --fetching salary of the given employee
9. SELECT salary INTO v\_salary
10. FROM employees WHERE employee\_id = p\_employee\_id;
11. --logic for tax percentage computation
12. IF v\_salary >=15000 THEN
13. v\_tax\_percentage := 15;
14. ELSIF v\_salary < 15000 AND v\_salary >= 8000 THEN
15. v\_tax\_percentage := 10;
16. ELSE
17. v\_tax\_percentage := 0;
18. END IF;
19. --formula for tax calculation
20. v\_tax\_amount:=v\_salary\*v\_tax\_percentage\*0.01;
21. DBMS\_OUTPUT.put\_line('Employee has to pay tax of Rs.'||v\_tax\_amount);
22. END;

**NOTE:**

Conditions in IF-THEN-ELSIF are mutually exclusive. It means, if any of the conditions evaluate to TRUE, the group of statements written inside that condition will get executed and rest of the conditions will be ignored.

While using this construct it is not necessary to write ELSE clause. You can ignore the ELSE clause if you don't have any default statements to be executed if none of the conditions evaluate to TRUE.

Highlights:

 Requirement 6: HR Manager wants to add a new job Finance Clerk (FI\_CLERK)

 Develop a PL/SQL program to add Finance Clerk, FI\_CLERK as a new job with minimum and maximum salary as 5000 and 10000 respectively. Add it to the database only if the same job ID does not exist, else display the message "FI\_CLERK already exists".

Demosteps:

1. DECLARE
2. v\_job\_ID VARCHAR2(10) := 'FI\_CLERK';
3. v\_job\_title VARCHAR2 (35) := 'Finance Clerk';
4. v\_min\_salary NUMBER (6) := 5000;
5. v\_max\_salary NUMBER (6) := 10000;
6. v\_count\_of\_jobs PLS\_INTEGER;
7. BEGIN
8. --Retrieving the number of jobs with same job\_id
9. */\*COUNT() is an aggregate funtion which will always return one row result,*
10. *so even if there are no matching records in the table it will return 0*
11. *and will not lead to an exception \*/*
12. SELECT COUNT (Job\_id)
13. INTO v\_count\_of\_jobs FROM jobs WHERE job\_id = v\_job\_ID;
15. IF (v\_count\_of\_jobs = 0)
16. THEN --if count is 0 indicate given job ID doesn't exist in jobs table
17. INSERT INTO Jobs
18. VALUES (v\_job\_ID,v\_job\_title,v\_min\_salary,v\_max\_salary);
19. ELSE --otherwise  it indicates given job ID already exist
20. DBMS\_OUTPUT.PUT\_LINE (v\_job\_title || ' already exists');
21. END IF;
22. END;

## ProblemStatement:

# ****Requirement 7:****An employee wants to update his/her phone number

Develop a PL/SQL program to update the phone number of John Chen, employee ID 110, as 515.124.5845.

## ProblemStatement:

# ****Requirement 8:**** HR Manager wants to update commission percentage of employees

Develop a PL/SQL program to update the commission percentage (Commission\_PCT) of Pat, employee ID 202, as 0.1, if he is a marketing rep (Job\_ID :MK\_REP).

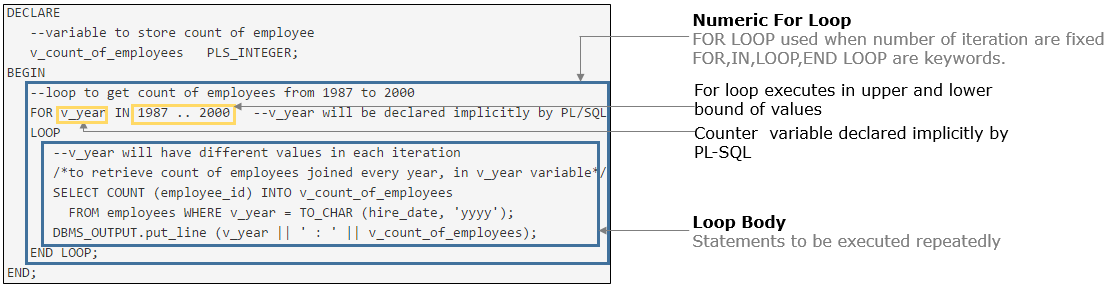
# ****Requirement 9:****HR manager wants a report of how many employees joined in every year from 1987 to 2000

# Execute the below existing program in SQL Developer and observe:

1. DECLARE
2. --variable to store count of employee
3. v\_count\_of\_employees   PLS\_INTEGER;
4. BEGIN
5. --loop to get count of employees from 1987 to 2000
6. FOR v\_year IN 1987 .. 2000   --v\_year will be declared implicitly by PL/SQL
7. LOOP
8. --v\_year will have different values in each iteration
9. */\*to retrieve count of employees joined every year, in v\_year variable\*/*
10. SELECT COUNT (employee\_id) INTO v\_count\_of\_employees
11. FROM employees WHERE v\_year = TO\_CHAR (hire\_date, 'yyyy');
12. DBMS\_OUTPUT.put\_line (v\_year || ' : ' || v\_count\_of\_employees);
13. END LOOP;
14. END;

**Requirement 9: Report how many employees joined in every year: Code anatomy**

**Usage of FOR loop**



# Requirement 9:  HR Manager wants a report of how many employees joined in every year

**Revisiting requirement 9**: HR manager wants the report having count of employees who joined InfoSpark every year, **from 1987 to current year.**

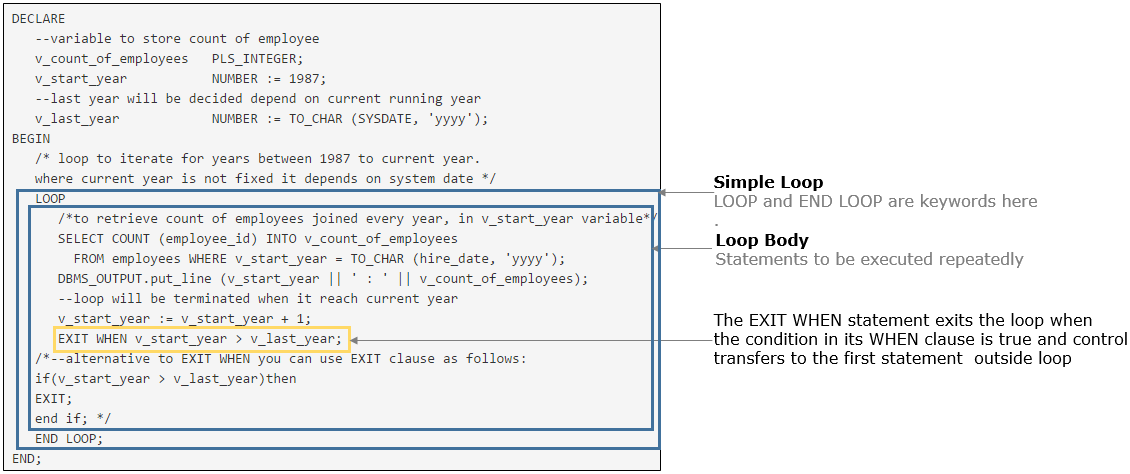
Execute the below existing program in SQL Developer and observe:

1. DECLARE
2. --variable to store count of employee
3. v\_count\_of\_employees   PLS\_INTEGER;
4. v\_start\_year           NUMBER := 1987;
5. --last year will be decided depend on current running year
6. v\_last\_year            NUMBER := TO\_CHAR (SYSDATE, 'yyyy');
7. BEGIN
8. */\* loop to iterate for years between 1987 to current year.*
9. *where current year is not fixed it depends on system date \*/*
10. LOOP
11. */\*to retrieve count of employees who joined every year, in v\_start\_year variable\*/*
12. SELECT COUNT (employee\_id) INTO v\_count\_of\_employees
13. FROM employees WHERE v\_start\_year = TO\_CHAR (hire\_date, 'yyyy');
14. DBMS\_OUTPUT.put\_line (v\_start\_year || ' : ' || v\_count\_of\_employees);
15. --loop will be terminated when it reach current year
16. v\_start\_year := v\_start\_year + 1;
17. EXIT WHEN v\_start\_year > v\_last\_year;
18. */\*--alternative to EXIT WHEN you can use EXIT clause as follows:*
19. *if(v\_start\_year > v\_last\_year)then*
20. *EXIT;*
21. *end if; \*/*
22. END LOOP;
23. END;

**EXIT statement** forces a loop to complete unconditionally. When an EXIT statement is encountered, the loop completes immediately and control passes to the next statement.

**EXIT-WHEN statement** lets a loop complete conditionally. When the EXIT statement is encountered, the condition in the WHEN clause is evaluated. If the condition is true, the loop completes and control passes to the next statement after the loop.

**Requirement 9: HR Manager wants a report of how many employees joined in every year: Code anatomy**



Loops are used to execute a block of code multiple times. Oracle provides three types of loop statements - Simple loop, WHILE loop and FOR loop.

| **SIMPLE  LOOP** | **WHILE LOOP** | **NUMERIC FOR LOOP** |
| --- | --- | --- |
| 1. BEGIN 2. **LOOP** 3. **STATEMENT 1;** 4. **.** 5. **.** 6. **STATEMENT N;** 7. **EXIT WHEN CONDITION;** 8. **END LOOP;** 9. END; | 1. BEGIN 2. **WHILE CONDITION** 3. **LOOP** 4. **STATEMENT 1;** 5. **.** 6. **.** 7. **STATEMENT N;** 8. **END LOOP;** 9. END; | 1. BEGIN 2. **FOR <loop\_counter> IN [REVERSE] <lower\_limit number> .. <upper\_limit number>** 3. **LOOP** 4. **STATEMENT 1;** 5. **.** 6. **.** 7. **STATEMENT N;** 8. **END LOOP;** 9. END; |
| Use this loop when   * You are not sure how many times you want the loop to execute * You want the loop body to run at least once | Use this loop when   * You are not sure how many times you want the loop to execute * You want to conditionally terminate the loop * You don't have to execute loop body even one time | Use this loop when   * You want to execute  a body of loop a fixed number of times and do not want to halt the looping prematurely   The body of the loop executes once for every integer value between the lower\_limit and upper\_limit values |

## ProblemStatement:

# Requirement 9: HR Manager wants a report of how many employees joined in every year

**Revisiting requirement 9:** HR Manager wants the report having count of employees who joined InfoSpark every year, **from 1987 to current year.**

**Hint :**Implement the same logic in the previous requirement using WHILE loop

# Requirement 10: HR Manager wants to add employee

Whenever a new employee is recruited to the company, HR manager should add the employee details to the database.

Execute the PL/SQL program to add Joy to the database, his details are:

EMPLOYEE\_ID : 900

FIRST\_NAME : Joy

LAST\_NAME : Bosh

EMAIL : Joy\_Bosh

HIRE\_DATE: Today's date

JOB\_ID: FI\_MGR

1. DECLARE
2. v\_first\_name VARCHAR2 (20) := 'Joy';
3. v\_last\_name  VARCHAR2 (25) := 'Bosh';
4. v\_email      VARCHAR2 (25) := 'Joy\_Bosh';
5. --hire date is today's date,retrieved from database time
6. v\_hire\_date   DATE          := SYSDATE;
7. v\_job\_id      VARCHAR2 (10) := 'FI\_MGR';
8. v\_employee\_id NUMBER (6)    :=900;
9. BEGIN
10. --employee Joy Bosh added to employees table
11. INSERT
12. INTO employees
13. (
14. EMPLOYEE\_ID,FIRST\_NAME,LAST\_NAME,EMAIL,HIRE\_DATE,JOB\_ID
15. )
16. VALUES
17. (
18. v\_employee\_id,v\_first\_name,v\_last\_name,v\_email,v\_hire\_date,v\_job\_id
19. );
20. END;

# Requirement 10: HR Manager wants to add employee

The company has recruited one more employee and when entering the below employee details into the system, HR Manager is getting an error.

EMPLOYEE\_ID : 901

FIRST\_NAME : Prince Christian Victor

LAST\_NAME : Taylor

EMAIL : PCTaylor

JOB\_ID: FI\_MGR

So a ticket is raised to the support team to add his details from the back end.

# Hence, you as part of the support team execute the below Insert statement and observe:

1. INSERT INTO employees
2. (
3. EMPLOYEE\_ID,FIRST\_NAME,LAST\_NAME,EMAIL,HIRE\_DATE,JOB\_ID
4. )
5. VALUES
6. (
7. 901,'Prince Christian Victor','Taylor','PCTaylor',SYSDATE,'FI\_MGR'
8. );

On executing the above insert statement, **you will get the exception:**

**SQL Error: ORA-12899: value too large for column "EMPLOYEES"."FIRST\_NAME" (actual: 23, maximum: 20)**

# Requirement 10: HR Manager wants to add employee

By analyzing the exception, you can observe a problem that the first name of the new employee has **23 characters** which is more than the column size of first name in the employees table.

1. --DESC[RIBE] command lists the column definitions for the specified table
2. DESC employees

So to accommodate his details in the table, increase the column size by executing the below ALTER command.

1. ALTER TABLE employees MODIFY first\_name VARCHAR2(30);

# ****Now, execute the INSERT statement once again to add Prince Christian Victor to the database:****

1. INSERT INTO employees
2. (
3. EMPLOYEE\_ID,FIRST\_NAME,LAST\_NAME,EMAIL,HIRE\_DATE,JOB\_ID
4. )
5. VALUES
6. (
7. 901,'Prince Christian Victor','Taylor','PCTaylor',SYSDATE,'FI\_MGR'
8. );

# Requirement 2: An employee can view the first name and phone number of other employees

Using the program developed for Requirement 2, retrieve and display the first name and phone number of the newly added employee, **Prince Christian Victor.**

1. DECLARE
2. v\_employee\_id NUMBER (6):= 901; -- assign the employee id of Prince Christian Victor
3. v\_first\_name VARCHAR2(20);
4. v\_phone\_number VARCHAR2(25);
5. BEGIN
7. --Values of Prince Christian Victor are retrieved and stored in the variables
8. SELECT first\_name, phone\_number
9. INTO v\_first\_name, v\_phone\_number
10. FROM employees where employee\_id= v\_employee\_id;
12. DBMS\_OUTPUT.PUT\_LINE ('Employee ID: ' || v\_employee\_id);
13. DBMS\_OUTPUT.PUT\_LINE ('First Name: ' || v\_first\_name);
14. DBMS\_OUTPUT.PUT\_LINE ('Contact Number: ' || v\_phone\_number);
15. END;

You will get "**ORA-06502: PL/SQL: numeric or value error: character string buffer too small** " exception, as the v\_first\_name variable is declared as VARCHAR2(20) which can accept only 20 characters. You can modify the declaration of v\_first\_name to fix it.

# Requirement 2: An employee can view the first name and phone number of other employees

Using the below modified program developed for Requirement 2, retrieve and display the first name and phone number of the newly added employee, **Prince Christian Victor.**

1. DECLARE
2. v\_employee\_id NUMBER (6):= 901; -- assign the employee id of Prince Christian Victor
3. v\_first\_name VARCHAR2(30); **--size of the variable is increase**d
4. v\_phone\_number VARCHAR2(25);
5. BEGIN
7. --Values of Prince Christian Victor are retrieved and stored in the variables
8. SELECT first\_name, phone\_number
9. INTO v\_first\_name, v\_phone\_number
10. FROM employees where employee\_id= v\_employee\_id;
12. DBMS\_OUTPUT.PUT\_LINE ('Employee ID: ' || v\_employee\_id);
13. DBMS\_OUTPUT.PUT\_LINE ('First Name: ' || v\_first\_name);
14. DBMS\_OUTPUT.PUT\_LINE ('Contact Number: ' || v\_phone\_number);
15. END;

But is this the only solution, should you keep modifying the PL/SQL programs whenever there is a change in column definition?

No, **Anchored declaration**helps to resolve this.

Highlights:

 Revisiting requirement 2: An employee wants to view the first name and phone number of other employees

 Develop a PL/SQL program to retrieve and display the first name and phone number of employee with employee ID 100, with the variables anchor declared.

Demosteps:

**Step 1:** Execute the below code in SQL Developer and observe the output.

1. DECLARE
2. v\_employee\_id number(5):= 100;
3. ***/\*v\_first\_name will be declared with the same data type and size***
4. ***of the first\_name column in the employees table \*/***
5. **v\_first\_name employees.first\_name%type;**
6. v\_phone\_number varchar2(20);
7. BEGIN
9. --Values of employee with id 100 are retrieved and stored in the variables
10. SELECT first\_name, phone\_number
11. INTO v\_first\_name, v\_phone\_number
12. FROM employees where employee\_id= v\_employee\_id;
14. DBMS\_OUTPUT.PUT\_LINE ('Employee ID: ' || v\_employee\_id);
15. DBMS\_OUTPUT.PUT\_LINE ('First Name: ' || v\_first\_name);
16. DBMS\_OUTPUT.PUT\_LINE ('Contact Number: ' || v\_phone\_number);
17. END;

We have explicitly specified the datatype for the variables declared so far. This approach of variable declarations is certainly valid.

But, as the size of first\_name column is changed to VARCHAR2(30) your existing PL/SQL programs may not work, if the size of the value (name) to be stored exceeds the size of variable declared to store that value.

One solution is to make the changes throughout the application and increase the size of your variables, but this is a tedious job.

So, for synchronization of variable datatype with datatype of database columns, **anchored declaration**of variables are more advisable.

# 

# Anchored Declaration:

It is used to set the datatype of your variable based on the datatype of an already defined variable or a column of a table.

PL/SQL offers two kinds of anchoring:

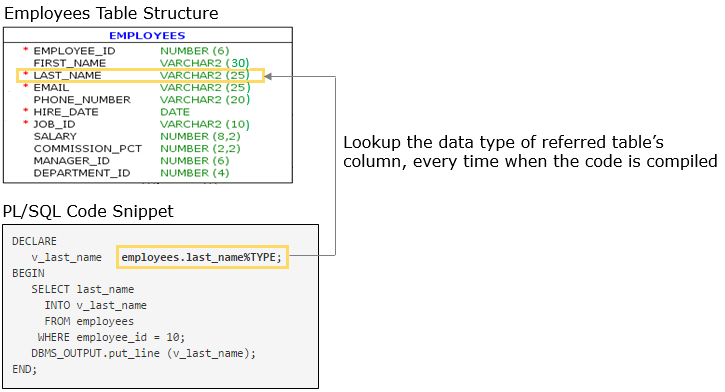
**Scalar anchoring** - Use the %TYPE attribute to define your variable based on a table's column or an already declared PL/SQL scalar variable.

**Record anchoring** - Use the %ROWTYPE attribute to define your record structure based on a table or a predefined PL/SQL explicit cursor.

It is used to declare variables that directly map to a column definition in the database. An anchored declaration includes a**fully-qualified column name**followed by the keyword **%TYPE.**

**Syntax:**

1. <variable\_name> <table\_name>.<column\_name>%TYPE [optional default value assignment]

****

**When variable is anchored to the column of a table, only datatype and size is referred, not the constraints.**

**For example, in the above code, last\_name column is constrained with not null. But, variable v\_last\_name  anchored to that column will not have any such constraint.**

## ProblemStatement:

# Requirement 2: An employee wants to view the details of other employees

**Revisiting requirement 2**: An employee wants to view the first name, email and phone number of other employees.

Develop a PL/SQL program to retrieve and display the first name, email and phone number of employee with employee ID 100, with the variables anchor declared.

**Hint:** Modify the code in the previous demo and anchor all the possible variables to the columns of Employees table

## ProblemStatement:

# Requirement 3: Manager wants to view the joining date of other employees

**Revisiting requirement 3:** Manager wants to view the joining date, in addition to first name, email, phone number  and joining date of other employees. Retrieve and display the details of employee with ID 104, with the variables anchor declared to Employees table.

Enhance the code to achieve the requirement.

1. DECLARE
2. v\_employee\_id employees.employee\_id%type := 104;
4. */\* Anchor declare the variables with appropriate columns of employees table*
5. *to store email id, joining date and department id\*/*
7. BEGIN
9. --Retrieve the values of 104 from the database using Select statement
11. DBMS\_OUTPUT.PUT\_LINE ('Employee ID: ' || v\_employee\_id);
12. -- Display the other details in similar format
14. END;

## ProblemStatement:

# Requirement 6: HR Manager wants to add a new job

**Revisiting requirement 6:** HR Manager wants to add a new job Advertising Director (ADV\_DIR).

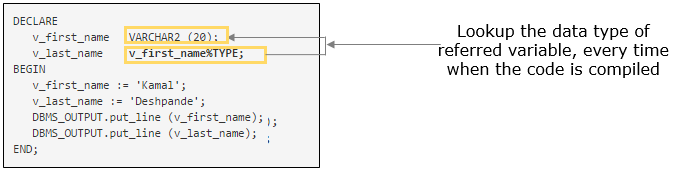
Develop a PL/SQL program to add Advertising Director, ADV\_DIR as a new job with minimum and maximum salary as 5000 and 10000 respectively. Add it to the database only if the same job ID does not exist, else display the message "ADV\_DIR already exists".

# Anchor declare the variables in below program:

1. DECLARE
2. **-- Anchor declare the below variables**
3. v\_job\_ID
4. v\_job\_title
5. v\_min\_salary
6. v\_max\_salary
8. **--v\_count\_of\_jobs is not used to store values from any table, so it can be declared statically**
9. v\_count\_of\_jobs PLS\_INTEGER;
10. BEGIN
11. --Retrieving the number of jobs with same job\_id
12. */\*COUNT() is an aggregate funtion which will always return one row result,*
13. *so even if there are no matching records in the table it will return 0*
14. *and will not lead to an exception \*/*
15. SELECT COUNT (Job\_id)
16. INTO v\_count\_of\_jobs FROM jobs WHERE job\_id = v\_job\_ID;
18. IF (v\_count\_of\_jobs = 0)
19. THEN --if count is 0 indicate given job ID doesn't exist in jobs table
20. INSERT INTO Jobs
21. VALUES (v\_job\_ID,v\_job\_title,v\_min\_salary,v\_max\_salary);
22. ELSE --otherwise it indicates given job ID already exist
23. DBMS\_OUTPUT.PUT\_LINE (v\_job\_title || ' already exists');
24. END IF;
25. END;

It can be possible that a set of variables hold values of same datatype. If all the variables are explicitly declared and when there is a change in datatype, you may need to change all the variable declarations.

Instead, you can have one variable explicitly declared or anchor declared to a column and have the rest of the variables anchor declared to this variable. Hence, in case of any change you need to modify only one variable's declaration.



**Execute the below code and observe:**

1. DECLARE
2. v\_max\_marks NUMBER(3) := 100;
3. **--variables anchor declared to v\_max\_marks,**
4. **--v\_maths\_marks and v\_physics\_marks will have the same datatype as v\_max\_marks**
5. v\_maths\_marks v\_max\_marks%type;
6. v\_physics\_marks v\_max\_marks%type;
7. BEGIN
8. v\_maths\_marks := 75;
9. v\_physics\_marks := 80;
10. DBMS\_OUTPUT.put\_line (v\_maths\_marks);
11. DBMS\_OUTPUT.put\_line (v\_physics\_marks);
12. END;

# Requirement 1: HR Manager wants to update the salary of employees

**Revisiting requirement 1:** HR Manager Wants To Update The Salary Of Employees

Based on the latest compensation review, the company has decided to give an annual hike of 10% to all its employees and an additional hike of 5% to its employees who have joined before '01-Jan-95'.   
HR manager wants to calculate the new salary of Alexander based on these rules, update the database and also display a report in the below format.

Name : <First\_Name> <Last\_Name>  
Date of Joining : <Hire\_Date>  
Current Salary : <Salary>  
Incremented Salary : <Calculated Salary>

**Variable to store the calculated salary can be anchored to an already declared variable v\_salary rather than anchoring it to the table.**

1. set SERVEROUTPUT ON;
2. DECLARE
3. -- Anchor declared variables to store the retrieved values of Alexander
4. v\_employee\_id employees.employee\_id%type:=103; --Alexander's employee ID is 103
5. v\_first\_name employees.first\_name%type;
6. v\_last\_name employees.last\_name%type;
7. v\_Date\_of\_Joining employees.hire\_date%type;
8. v\_salary employees.salary%type;
10. **/\*Variable to store the calculated salary,**
11. **this can be anchored to already declared variable v\_salary rather than anchoring it to the table\*/**
12. **v\_incremented\_salary v\_salary%type;**
13. BEGIN
14. -- Query to fetch the values of Alexander from DB
15. SELECT first\_name,last\_name,hire\_date,salary
16. INTO v\_first\_name,v\_last\_name,v\_Date\_of\_Joining,v\_salary
17. FROM employees WHERE employee\_id = v\_employee\_id;
19. -- Checking if the hire date is before 01-Jan-95
20. IF(v\_Date\_of\_Joining < '01-Jan-95') THEN
21. -- Calculating new salary with 10+5 % hike
22. v\_incremented\_salary := v\_salary \* 1.15;
23. ELSE
24. -- Calculating new salary with 10% hike
25. v\_incremented\_salary := v\_salary \* 1.1;
26. END IF;
28. -- Updating the salary of Alexander in the DB
29. UPDATE employees SET salary = v\_incremented\_salary WHERE employee\_id = v\_employee\_id;
31. --Displaying the report
32. DBMS\_OUTPUT.put\_line ('Name : ' || v\_First\_Name ||' '|| v\_Last\_Name);
33. DBMS\_OUTPUT.put\_line ('Date of Joining : ' || v\_Date\_of\_Joining);
34. DBMS\_OUTPUT.put\_line ('Current Salary : ' || v\_Salary);
35. DBMS\_OUTPUT.put\_line ('Incremented Salary : ' || v\_incremented\_salary);
36. END;

# ****Requirement 11:****HR Manager wants to view employee details

HR manager wants to view all the details of an employee Joy Bosh (employee ID: 900), who has recently joined.

Develop a PL/SQL program to display all the details of an employee Joy Bosh.

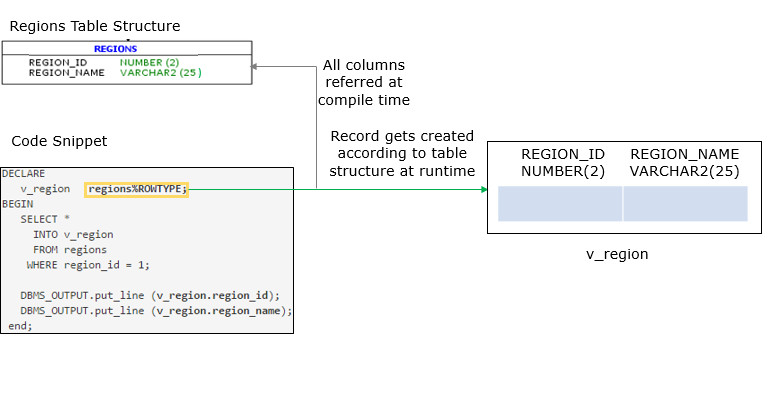
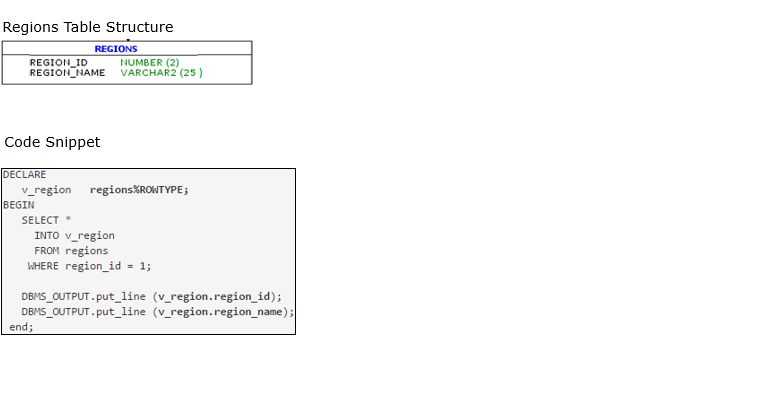
1. DECLARE
2. ***/\* %type anchors to a column,likewise here we are creating a variable***
3. ***by anchoring it to entire row of employees table using %rowtype \*/***
4. **rec\_emp\_detail employees%rowtype;**
5. v\_employee\_id employees.employee\_id%type:=900;
6. BEGIN
7. */\*Fetching entire row of employees table into %rowtype anchored variable (Record)\*/*
8. SELECT  \* INTO  rec\_emp\_detail FROM employees WHERE employee\_id = v\_employee\_id;
9. */\*Individual fields in fetched record rec\_emp\_detail, can be retrieved*
10. *using record\_name.field\_name as follows \*/*
11. */\*Names of the fields in the %rowtype record will be same as column name in table\*/*
12. DBMS\_OUTPUT.PUT\_LINE ('EMPLOYEE ID=' || rec\_emp\_detail.employee\_id);
13. DBMS\_OUTPUT.PUT\_LINE ('FIRST NAME=' || rec\_emp\_detail.first\_name);
14. DBMS\_OUTPUT.PUT\_LINE ('LAST NAME=' || rec\_emp\_detail.last\_name);
15. DBMS\_OUTPUT.PUT\_LINE ('EMAIL=' || rec\_emp\_detail.email);
16. DBMS\_OUTPUT.PUT\_LINE ('PHONE NUMBER=' || rec\_emp\_detail.phone\_number);
17. DBMS\_OUTPUT.PUT\_LINE ('HIRE DATE=' || rec\_emp\_detail.hire\_date);
18. DBMS\_OUTPUT.PUT\_LINE ('JOB ID=' || rec\_emp\_detail.job\_id);
19. DBMS\_OUTPUT.PUT\_LINE ('SALARY=' || rec\_emp\_detail.salary);
20. DBMS\_OUTPUT.PUT\_LINE ('COMMISSION PERCENTAGE=' || rec\_emp\_detail.commission\_pct);
21. DBMS\_OUTPUT.PUT\_LINE ('DEPARTMENT ID=' || rec\_emp\_detail.department\_id);
22. END;

# ****Requirement 11:****HR Manager wants to view employee details

HR manager wants to view all the details of an employee Joy Bosh (employee ID: 900), who has recently joined.

Develop a PL/SQL program to display all the details of an employee Joy Bosh.

1. DECLARE
2. ***/\* %type anchors to a column,likewise here we are creating a variable***
3. ***by anchoring it to entire row of employees table using %rowtype \*/***
4. **rec\_emp\_detail employees%rowtype;**
5. v\_employee\_id employees.employee\_id%type:=900;
6. BEGIN
7. */\*Fetching entire row of employees table into %rowtype anchored variable (Record)\*/*
8. SELECT  \* INTO  rec\_emp\_detail FROM employees WHERE employee\_id = v\_employee\_id;
9. */\*Individual fields in fetched record rec\_emp\_detail, can be retrieved*
10. *using record\_name.field\_name as follows \*/*
11. */\*Names of the fields in the %rowtype record will be same as column name in table\*/*
12. DBMS\_OUTPUT.PUT\_LINE ('EMPLOYEE ID=' || rec\_emp\_detail.employee\_id);
13. DBMS\_OUTPUT.PUT\_LINE ('FIRST NAME=' || rec\_emp\_detail.first\_name);
14. DBMS\_OUTPUT.PUT\_LINE ('LAST NAME=' || rec\_emp\_detail.last\_name);
15. DBMS\_OUTPUT.PUT\_LINE ('EMAIL=' || rec\_emp\_detail.email);
16. DBMS\_OUTPUT.PUT\_LINE ('PHONE NUMBER=' || rec\_emp\_detail.phone\_number);
17. DBMS\_OUTPUT.PUT\_LINE ('HIRE DATE=' || rec\_emp\_detail.hire\_date);
18. DBMS\_OUTPUT.PUT\_LINE ('JOB ID=' || rec\_emp\_detail.job\_id);
19. DBMS\_OUTPUT.PUT\_LINE ('SALARY=' || rec\_emp\_detail.salary);
20. DBMS\_OUTPUT.PUT\_LINE ('COMMISSION PERCENTAGE=' || rec\_emp\_detail.commission\_pct);
21. DBMS\_OUTPUT.PUT\_LINE ('DEPARTMENT ID=' || rec\_emp\_detail.department\_id);
22. END;



## https://academy.onwingspan.com/common-content-store/Shared/Shared/Public/lex_13675291041078530000_shared/web-hosted/assets/plsql_anchor_slider3_26_new3.PNGhttps://academy.onwingspan.com/common-content-store/Shared/Shared/Public/lex_13675291041078530000_shared/web-hosted/assets/plsql_anchor_slider4_27_new3.PNGhttps://academy.onwingspan.com/common-content-store/Shared/Shared/Public/lex_13675291041078530000_shared/web-hosted/assets/plsql_anchor_slider5_28_new3.PNGhttps://academy.onwingspan.com/common-content-store/Shared/Shared/Public/lex_13675291041078530000_shared/web-hosted/assets/plsql_anchor_slider6_29_new3.PNGProblemStatement:

# Requirement 12: HR Manager wants to view all the details of department

HR Manager wants to view all the details of department ID 10

Develop PL/SQL program to display all the details of department.

1. DECLARE
2. v\_department\_id departments.department\_id%type:=10;
3. */\*declare a variable anchored to the departments table and*
4. *name as rec\_dept\_details ,to fetch following details\*/*
5. BEGIN
6. --retrieve and store the details of deparment 10 in %rowtype variable
7. --display all the retrieved details
8. END;

## ProblemStatement:

# Requirement 13: HR Manager wants to add new country

InfoSpark decided to expand their business in two more countries Singapore (SG) and Sri Lanka (LK) in Asia region.

Write a PL/SQL block to add these countries to DB. Check before adding country, whether they already exist. If exists, then don't add them to DB and display appropriate message.

**Hint**: Insert records into Countries table.

1. DECLARE
2. v\_region\_name regions.region\_name%TYPE := 'Asia';
3. --get the region ID from regions table
4. v\_region\_id regions.region\_id%TYPE;
5. --change the values of folloeing variables to add other countries
6. v\_country\_name countries.country\_name%TYPE := 'Singapore';
7. v\_country\_id countries.country\_id%TYPE := 'SG';
8. BEGIN

11. --write the logic to add the countries

14. end;

## ProblemStatement:

# Requirement 14: HR Manager wants to know how many employees are working on a particular job

HR manager needs a report that shows how many employees are working on a particular job.

Write a PL/SQL code to get report for 'Programmer' job.

Before generating report, given job should be checked whether it is valid or not. If not, display appropriate message.

1. DECLARE
2. v\_job\_id jobs.job\_id%TYPE;
3. v\_job\_title jobs.job\_title%TYPE := 'Programmer';
4. v\_no\_of\_emp NUMBER (3) := 0;
5. BEGIN
7. --write the logic to get count of employees working on given job title
9. end;

## ProblemStatement:

# Requirement 15: Employee wants to know whether he/she is eligible for promotion or not

Develop a PL/SQL program to check eligibility of David Austin, employee ID 105, for promotion. Employee should have completed at least 2 years with InfoSpark to be eligible for promotion.

## ProblemStatement:

# Requirement 16: HR Manager wants to update the manager of a department

HR Manager wants to update the manager details of a department.

Develop a PL/SQL program to make David Austin, employee ID 105, as the manager of IT department.

Business Rule: 105 should be a valid employee and should also belong to IT department.

## ProblemStatement:

# ****Revisiting requirement 8:**** HR Manager wants to update the commission percentage of employees

Develop a PL/SQL program to update the commission percentage(Commission\_PCT) of Pat, employee ID 202, as 0.1, if he is a marketing rep (Job\_ID :MK\_REP). Also calculate and display the total salary of Pat.

**Hint :** Total Salary = Salary \* ( 1+ Commission\_PCT)

# Requirement 10: HR Manager wants to add employee

Revisiting Requirement 10, whenever a new employee is recruited to the company, HR manager should add the employee's details to the database. Certain validations need to be performed, before allocating employee to particular department and job.

Check if department name exists or not. If exists, get department ID from Departments table.

Check if job title exists or not. If exists, get job ID from Jobs table.

Execute the PL/SQL program to add Steve to the database, his details are

EMPLOYEE\_ID : 902

FIRST\_NAME : Steve

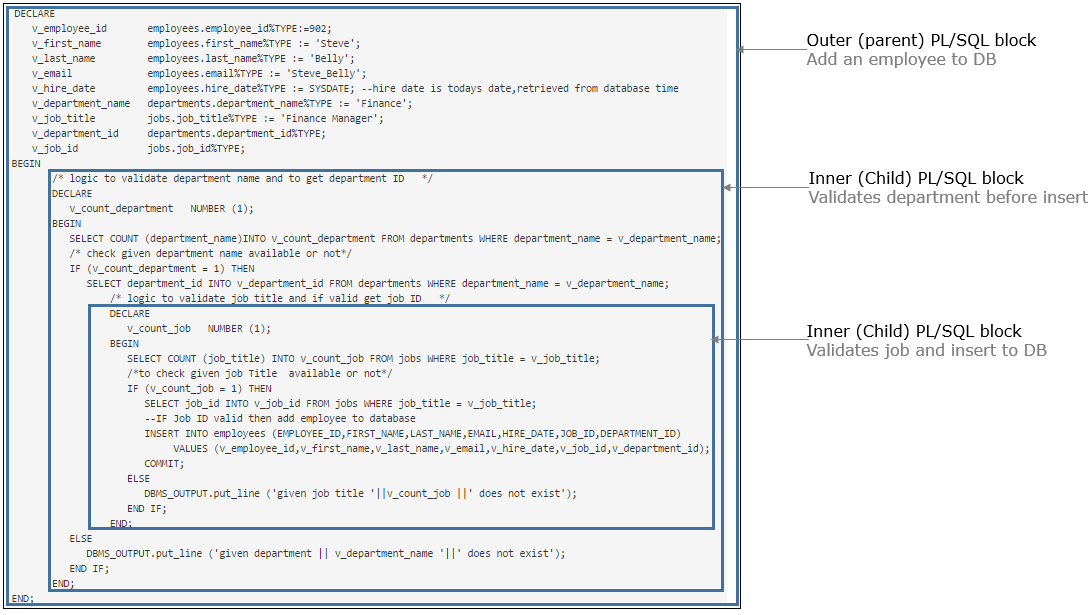
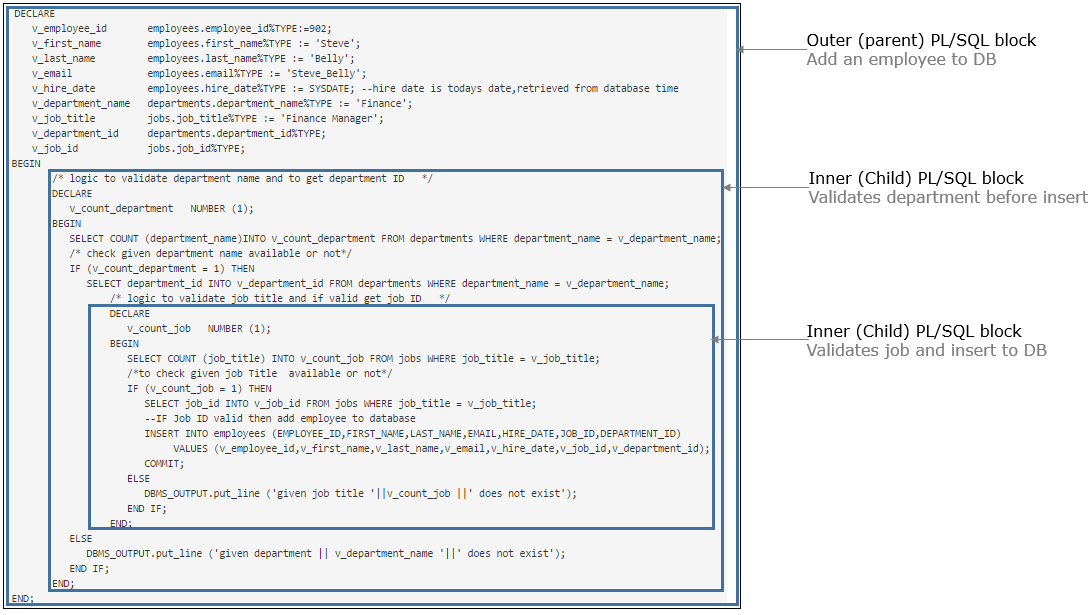
LAST\_NAME : Belly

EMAIL : Steve\_Belly

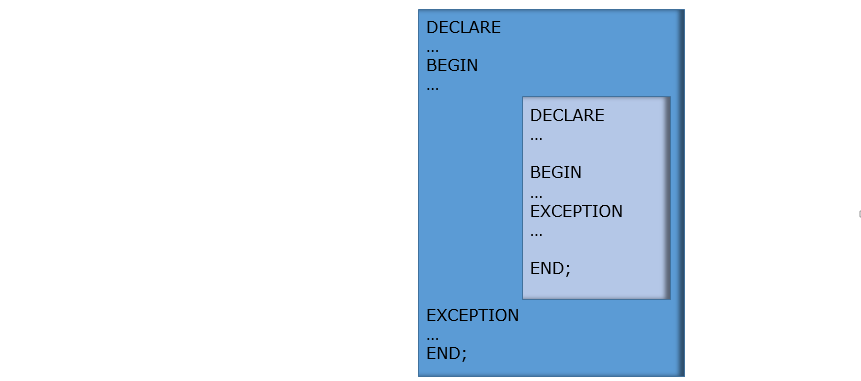
HIRE\_DATE: Today's date

JOB\_TITLE : Finance Manager

DEPARTMENT\_NAME : Finance

1. DECLARE
2. v\_employee\_id       employees.employee\_id%TYPE:=902;
3. v\_first\_name        employees.first\_name%TYPE := 'Steve';
4. v\_last\_name         employees.last\_name%TYPE := 'Belly';
5. v\_email             employees.email%TYPE := 'Steve\_Belly';
6. v\_hire\_date         employees.hire\_date%TYPE := SYSDATE; --hire date is todays date,retrieved from database time
7. v\_department\_name   departments.department\_name%TYPE := 'Finance';
8. v\_job\_title         jobs.job\_title%TYPE := 'Finance Manager';
9. v\_department\_id     departments.department\_id%TYPE;
10. v\_job\_id            jobs.job\_id%TYPE;
11. BEGIN
12. */\* logic to validate department name and to get department ID   \*/*
13. DECLARE
14. v\_count\_department   NUMBER (1);
15. BEGIN
16. SELECT COUNT (department\_name)INTO v\_count\_department FROM departments WHERE department\_name = v\_department\_name;
17. */\* check given department name available or not\*/*
18. IF (v\_count\_department = 1) THEN
19. SELECT department\_id INTO v\_department\_id FROM departments WHERE department\_name = v\_department\_name;
20. */\* logic to validate job title and if valid get job ID   \*/*
21. DECLARE
22. v\_count\_job   NUMBER (1);
23. BEGIN
24. SELECT COUNT (job\_title) INTO v\_count\_job FROM jobs WHERE job\_title = v\_job\_title;
25. */\*to check given job Title  available or not\*/*
26. IF (v\_count\_job = 1) THEN
27. SELECT job\_id INTO v\_job\_id FROM jobs WHERE job\_title = v\_job\_title;
28. --IF Job ID valid then add employee to database
29. INSERT INTO employees (EMPLOYEE\_ID,FIRST\_NAME,LAST\_NAME,EMAIL,HIRE\_DATE,JOB\_ID,DEPARTMENT\_ID)
30. VALUES (v\_employee\_id,v\_first\_name,v\_last\_name,v\_email,v\_hire\_date,v\_job\_id,v\_department\_id);
31. COMMIT;
32. ELSE
33. DBMS\_OUTPUT.put\_line ('given job title '||v\_job\_title ||' does not exist');
34. END IF;
35. END;
36. ELSE
37. DBMS\_OUTPUT.put\_line ('given department || v\_department\_name '||' does not exist');
38. END IF;
39. END;
40. END;
41. **Requirement 10: HR Manager wants to add employee: Code anatomy**
42. To make code modular and more readable, you can break a PL/SQL code block into smaller blocks of code having logically related functionality to support different business requirements
44. 
45. **Requirement 10: HR Manager wants to add employee: Code anatomy**
46. To make code modular and more readable, you can break a PL/SQL code block into smaller blocks of code having logically related functionality to support different business requirements
48. 

In case your executable section has code for many logically related functionalities to support multiple business requirements, you can break it down into smaller blocks that are nested one inside the other, hence making the code modular and easier to manage.  
PL/SQL supports having blocks declared inside other PL/SQL blocks. Blocks can be nested either in the execution part or in the exception handling part. The diagram shows outer (parent) block and a nested (child) block.



# Execute the below existing program in SQL Developer and observe:

1. DECLARE
2. v\_first\_name employees.first\_name%TYPE;
3. BEGIN
4. SELECT first\_name INTO v\_first\_name FROM employees WHERE employee\_id = 100;
5. DECLARE
6. v\_dept\_id employees.department\_id%TYPE;
7. BEGIN
8. SELECT department\_id INTO v\_dept\_id FROM employees WHERE employee\_id=100;
9. DBMS\_OUTPUT.put\_line ('in inner block -->first name: ' || v\_first\_name);
10. DBMS\_OUTPUT.put\_line ('in inner block-->department ID: ' || v\_dept\_id);
11. END;
12. DBMS\_OUTPUT.put\_line ('in outer block -->first name: ' || v\_first\_name);
13. --DBMS\_OUTPUT.put\_line ('in outer block-->department ID: ' || v\_dept\_id);
14. END;

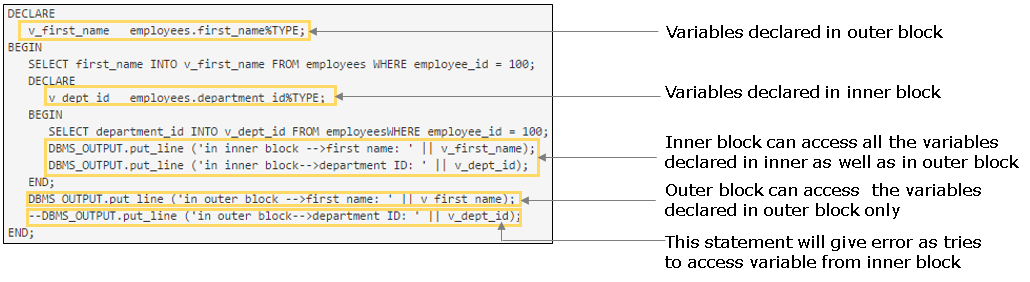
# 

# Now try out by un-commenting the line in above code and execute program in SQL Developer and observe the result:

You will get an error saying "PLS-00201: identifier 'V\_DEPT\_ID' must be declared"

The scope of a variable is that region of a program unit from which you can reference the variable. PL/SQL variables declared in the declare section of a block can be referenced till the end of that block. When block is completed, the memory used to store the value of this variable will get released and then you can't access them.

**Code Anatomy of previous PL/SQL code:**



Variables declared in a PL/SQL block can be accessed till the end of same block. But in case of nested blocks, PL/SQL gives the first preference to the variable inside its own block.

 PL/SQL Code Snippet :

1. */\*Qualifying Identifier for outer block\*/*
2. <<outer\_block>>
3. DECLARE
4. v\_tax\_per NUMBER;
5. BEGIN
6. v\_tax\_per := 7;
7. */\*Qualifying Identifier for inner block\*/*
8. <<inner\_block>>
9. DECLARE
10. v\_tax\_per NUMBER;
11. BEGIN
12. v\_tax\_per := 3;
13. */\*though variable declared in inner and outer block,*
14. *variables can be differentiated using Qualifying Identifier\*/*
15. DBMS\_OUTPUT.PUT\_LINE('in inner block v\_tax\_per: '||outer\_block.v\_tax\_per);
16. DBMS\_OUTPUT.PUT\_LINE('in inner block v\_tax\_per: '||inner\_block.v\_tax\_per);
17. END;
18. */\* we can not access the variable beyond the scope, even with the Qualifying Identifier\*/*
19. END;

In the above PL/SQL code snippet we have variables with the same name in both, outer and inner block.You can use a qualified name (block label and variable name separated by dot notation) to reference the variables from outer block, inside inner block. But if it is out of scope, it can't be accessed, even with the qualifying identifier.

In PL/SQL, a runtime error is called an exception.

An exception is normally raised in the executable section of the PL/SQL block. When an exception is raised, execution of the current block is terminated. The statements present after the error statement will not be executed. In this scenario, you have two options: to handle error within the same PL/SQL block or not to handle error within the PL/SQL block.

1. If the exception is handled, then the actions specified in the exception handler section is executed and PL/SQL block terminates successfully
2. If exception is not handled, then the block terminates with failure and the exception propagates to the calling environment

|  |  |
| --- | --- |
| https://academy.onwingspan.com/common-content-store/Shared/Shared/Public/lex_5445727406559867000_shared/web-hosted/assets/PLSQL_PLSQL_EXCEPTION_1_26Sep16_1227.JPG | https://academy.onwingspan.com/common-content-store/Shared/Shared/Public/lex_5445727406559867000_shared/web-hosted/assets/PLSQL_PLSQL_EXCEPTION_2_26Sep16_1227.JPG |

# Requirement 2: An employee wants to view the first name and phone number of other employees

**Revisiting requirement 2** to retrieve and display the first name and phone number of employee with employee ID 21. Now in addition, you need to display a message if the employee is not present.

# Execute the below existing program in SQL Developer and observe:

1. DECLARE
2. v\_employee\_id employees.employee\_id%TYPE := 21;
3. v\_first\_name employees.first\_name%TYPE;
4. v\_phone\_number employees.phone\_number%TYPE;
5. BEGIN
6. --Values of employee with id 21 are retrieved and stored in the variables
7. SELECT first\_name,
8. phone\_number
9. INTO v\_first\_name,
10. v\_phone\_number
11. FROM employees
12. WHERE employee\_id = v\_employee\_id;
13. DBMS\_OUTPUT.PUT\_LINE ('Employee ID: ' || v\_employee\_id);
14. DBMS\_OUTPUT.PUT\_LINE ('First Name: ' || v\_first\_name);
15. DBMS\_OUTPUT.PUT\_LINE ('Contact Number: ' || v\_phone\_number);
16. END;

You can observe that an **exception**'**ORA-01403: no data found**' is raised since there is no employee with the employee ID 21, and the program terminates unsuccessfully, hence the raised exception should be handled.

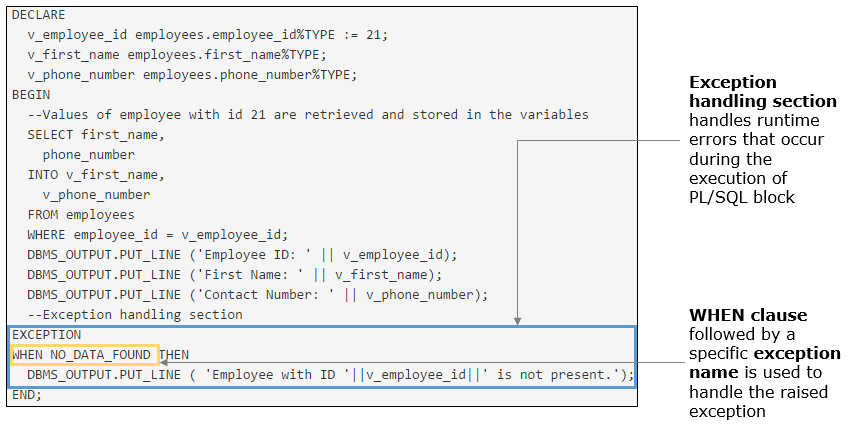
# 

# Requirement 2: An employee wants to view the first name and phone number of other employees

# Execute the below program in SQL Developer for handling the raised exception:

1. DECLARE
2. v\_employee\_id employees.employee\_id%TYPE := 21;
3. v\_first\_name employees.first\_name%TYPE;
4. v\_phone\_number employees.phone\_number%TYPE;
5. BEGIN
6. --Values of employee with id 21 are retrieved and stored in the variables
7. SELECT first\_name,
8. phone\_number
9. INTO v\_first\_name,
10. v\_phone\_number
11. FROM employees
12. WHERE employee\_id = v\_employee\_id;
13. DBMS\_OUTPUT.PUT\_LINE ('Employee ID: ' || v\_employee\_id);
14. DBMS\_OUTPUT.PUT\_LINE ('First Name: ' || v\_first\_name);
15. DBMS\_OUTPUT.PUT\_LINE ('Contact Number: ' || v\_phone\_number);
16. --Exception handling section
17. **EXCEPTION**
18. **WHEN NO\_DATA\_FOUND THEN**
19. **DBMS\_OUTPUT.PUT\_LINE ( 'Employee with ID '||v\_employee\_id||' is not present.');**
20. END;

**Requirement 2: An employee wants to view the first name and phone number of other employees: Code anatomy**

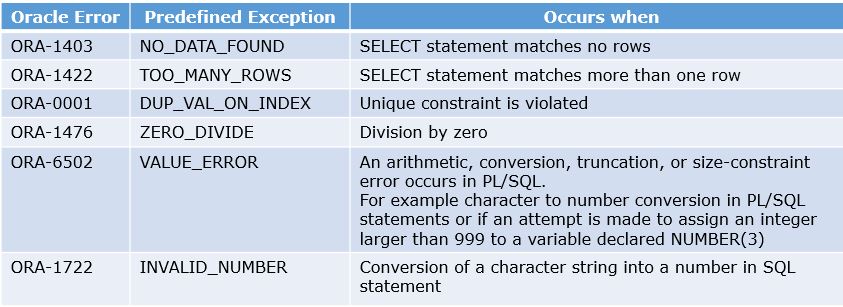


In previous requirement you observed that an exception 'no data found', which is a predefined exception, is handled using the exception name and the program terminates successfully.

Predefined exceptions are those exceptions that are

1. Predefined in Oracle server with an exception name and an Oracle error code
2. They are raised implicitly (automatically) by Oracle server
3. Handled using their standard name

The below table lists some of the names and error codes of the predefined exceptions. Check here for other [predefined exceptions and details](https://docs.oracle.com/cd/B28359_01/appdev.111/b28370/errors.htm#i9355)



## ProblemStatement:

# ****Requirement 17****: Manager wants to check the job history of an employee

Develop a PL/SQL program to view the employee ID, start date, end date, job ID of employee with employee ID 103 and display appropriate message if the employee doesn't have job history.

**Note:** If more than one records are available in job\_history table for the same employee then get the latest one.

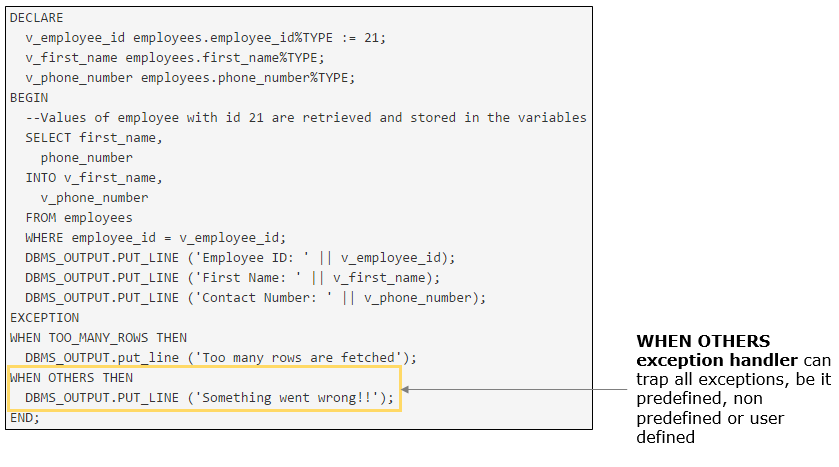
# Requirement 2: An employee wants to view the first name and phone number of other employees

**Revisiting Requirement 2** to retrieve and display the first name and phone number of employee with employee ID 21.

# Modify the code as shown below and execute the below program in SQL Developer and observe:

1. DECLARE
2. v\_employee\_id employees.employee\_id%TYPE := 21;
3. v\_first\_name employees.first\_name%TYPE;
4. v\_phone\_number employees.phone\_number%TYPE;
5. BEGIN
6. --Values of employee with id 21 are retrieved and stored in the variables
7. SELECT first\_name,
8. phone\_number
9. INTO v\_first\_name,
10. v\_phone\_number
11. FROM employees
12. WHERE employee\_id = v\_employee\_id;
13. DBMS\_OUTPUT.PUT\_LINE ('Employee ID: ' || v\_employee\_id);
14. DBMS\_OUTPUT.PUT\_LINE ('First Name: ' || v\_first\_name);
15. DBMS\_OUTPUT.PUT\_LINE ('Contact Number: ' || v\_phone\_number);
16. EXCEPTION
17. WHEN TOO\_MANY\_ROWS THEN
18. DBMS\_OUTPUT.put\_line ('Too many rows are fetched');
19. **WHEN OTHERS THEN**
20. **DBMS\_OUTPUT.PUT\_LINE ('Something went wrong!!');**
21. END;

**Requirement 2: An employee wants to view the first name and phone number of other employees: Code anatomy**



It is not possible for you to detect all the possible errors that can occur in your code and handle it. This can be handled using **WHEN OTHERS exception handler**.

1. It should be the last clause, so that all previous handlers will be scanned first
2. It handles the exceptions which are not handled by any other WHEN clauses defined in the exception section
3. Only one WHEN OTHERS clause is allowed in a PL/SQL block
4. It should be placed at the outermost block so that no errors go undetected or else the error will propagate to the calling environment
   1. DECLARE
   2. <declaration section>
   3. BEGIN
   4. <executable command(s)>
   5. EXCEPTION
   6. WHEN exception1 [OR exception2 . . .] THEN
   7. statement1;
   8. statement2;
   9. . . .
   10. [WHEN exception3 [OR exception4 . . .] THEN
   11. statement3;
   12. statement4;
   13. . . .]
   14. **[WHEN OTHERS THEN**
   15. **statement5;**
   16. **statement6;**
   17. **. . .]**
   18. END;

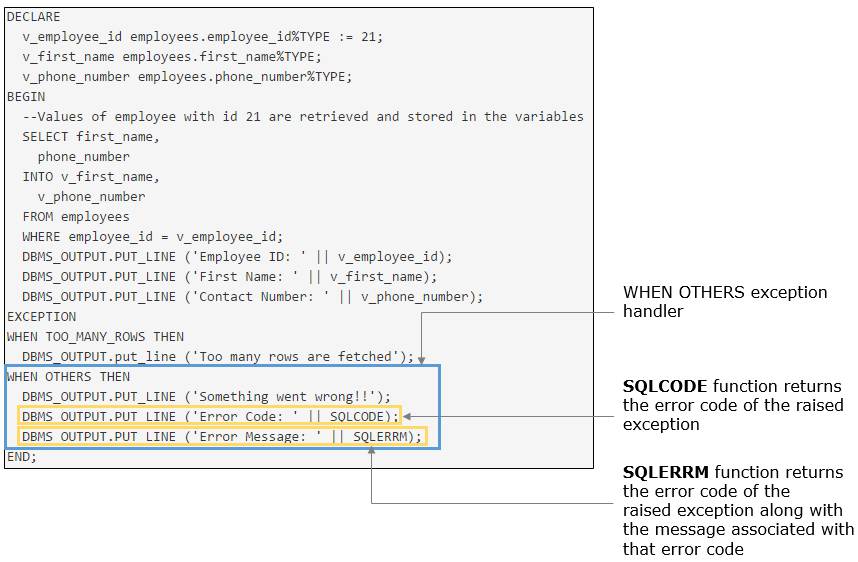
# Requirement 2: An employee wants to view the first name and phone number of other employees

**Revisiting requirement 2** to retrieve and display the first name and phone number of employee with employee ID 21.

# Modify the code as shown below and execute the below program in SQL Developer and observe:

1. DECLARE
2. v\_employee\_id employees.employee\_id%TYPE := 21;
3. v\_first\_name employees.first\_name%TYPE;
4. v\_phone\_number employees.phone\_number%TYPE;
5. BEGIN
6. --Values of employee with id 21 are retrieved and stored in the variables
7. SELECT first\_name,
8. phone\_number
9. INTO v\_first\_name,
10. v\_phone\_number
11. FROM employees
12. WHERE employee\_id = v\_employee\_id;
13. DBMS\_OUTPUT.PUT\_LINE ('Employee ID: ' || v\_employee\_id);
14. DBMS\_OUTPUT.PUT\_LINE ('First Name: ' || v\_first\_name);
15. DBMS\_OUTPUT.PUT\_LINE ('Contact Number: ' || v\_phone\_number);
16. EXCEPTION
17. WHEN TOO\_MANY\_ROWS THEN
18. DBMS\_OUTPUT.put\_line ('Too many rows are fetched');
19. WHEN OTHERS THEN
20. DBMS\_OUTPUT.PUT\_LINE ('Something went wrong!!');
21. **DBMS\_OUTPUT.PUT\_LINE ('Error Code: ' || SQLCODE);**
22. **DBMS\_OUTPUT.PUT\_LINE ('Error Message: ' || SQLERRM);**
23. END;

**Requirement 2: An employee wants to view the first name and phone number of other employees: Code anatomy**



Before you had handled the raised exception using only WHEN OTHERS exception handler but you did not know the reason behind the raised exception. You can get to know the raised exception details using **SQLCODE**and **SQLERRM**as in the previous requirement.

**SQLCODE**function returns the error code of the raised exception

**SQLERRM**function returns the error code of the raised exception along with the message associated with that error code

Check here for more details on [SQLCODE](https://docs.oracle.com/cd/B28359_01/appdev.111/b28370/sqlcode_function.htm) and [SQLERRM](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/sqlerrm_function.htm)

Highlights:

 Revisiting requirement 6: HR Manager wants to add a new job

 Develop a PL/SQL program to add a new job with details, job ID 'FI\_MGR' and job title 'Finance Manager', display appropriate message if the job ID is already present and include WHEN OTHERS exception handler.

Demosteps:

Step 1: Execute the below code in SQL Developer and observe the output.

1. DECLARE
2. v\_job\_id jobs.job\_id%TYPE := 'FI\_MGR';
3. v\_job\_title jobs.job\_title%TYPE := 'Finance Manager';
4. BEGIN
5. --inserting new job details
6. INSERT INTO jobs (job\_id,job\_title) VALUES(v\_job\_id,v\_job\_title);
7. --Exception handling section
8. EXCEPTION
9. WHEN DUP\_VAL\_ON\_INDEX THEN
10. DBMS\_OUTPUT.PUT\_LINE ('Job ID '||v\_job\_id||' already exists');
11. WHEN OTHERS THEN
12. DBMS\_OUTPUT.PUT\_LINE ('Something went wrong!!');
13. DBMS\_OUTPUT.PUT\_LINE ('Error Code: ' || SQLCODE);
14. DBMS\_OUTPUT.PUT\_LINE ('Error Message: ' || SQLERRM);
15. END;

In the above code, any exceptions other than duplicate value on index is handled by WHEN OTHERS exception handler.

## ProblemStatement:

# ****Revisiting requirement 17:**** Manager wants to check the job history of an employee

Develop a PL/SQL program to view the employee ID, start date, end date, job ID of employee with employee ID 103 and handle exceptions, if any, with appropriate message and include WHEN OTHERS exception handler.

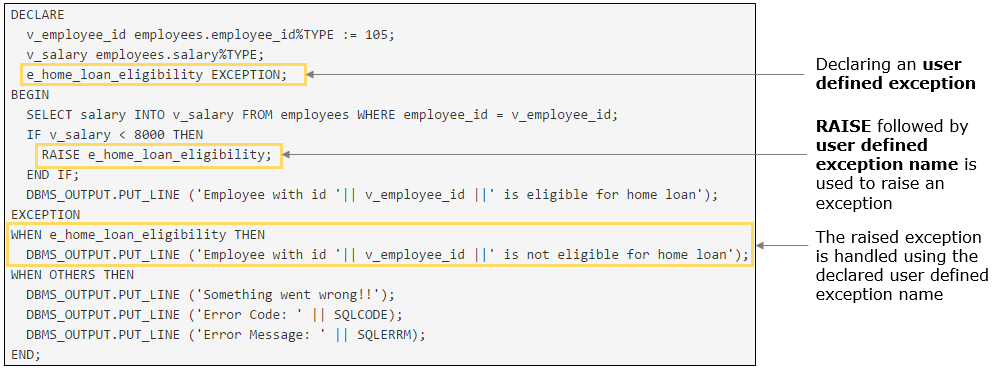
# Requirement 18: Employee wants to know whether he/she is eligible for home loan

Employee with the employee ID 105 wants to know whether he/she is eligible for home loan. An employee is eligible for home loan only if his/her salary is greater than or equal to 8000. If not eligible, raise an exception and handle with appropriate message.

# Execute the below existing program in SQL Developer and observe:

1. DECLARE
2. v\_employee\_id employees.employee\_id%TYPE := 105;
3. v\_salary employees.salary%TYPE;
4. e\_home\_loan\_eligibility EXCEPTION;
5. BEGIN
6. SELECT salary INTO v\_salary FROM employees WHERE employee\_id = v\_employee\_id;
7. **IF v\_salary < 8000 THEN**
8. **RAISE e\_home\_loan\_eligibility;**
9. **END IF;**
10. DBMS\_OUTPUT.PUT\_LINE ('Employee with id '|| v\_employee\_id ||' is eligible for home loan');
11. EXCEPTION
12. **WHEN e\_home\_loan\_eligibility THEN**
13. **DBMS\_OUTPUT.PUT\_LINE ('Employee with id '|| v\_employee\_id ||' is not eligible for home loan');**
14. WHEN OTHERS THEN
15. DBMS\_OUTPUT.PUT\_LINE ('Something went wrong!!');
16. DBMS\_OUTPUT.PUT\_LINE ('Error Code: ' || SQLCODE);
17. DBMS\_OUTPUT.PUT\_LINE ('Error Message: ' || SQLERRM);
18. END;

**Requirement 18 : Employee wants to know whether he/she is eligible for home loan: Code anatomy**



In the previous requirement you have observed that an exception with user defined name 'e\_home\_loan\_eligibility' is declared, raised and handled using the same name. Such exceptions are called as**User-defined Exception**.

Apart from the available exceptions, you can define your exception **based on the business requirement**.

For example: In a banking application, when a user is withdrawing money and the balance available is less than the withdrawal amount, then there is no predefined exception to notify the user. Hence, a user defined exception can be thrown and handled with a message saying 'Insufficient Balance' to the user.

User defined exceptions have to be

1. Declared in the declaration section
2. Explicitly raised using RAISE statement
3. Handled in exception handling section using the name
4. DECLARE
5. e\_user\_error EXCEPTION; --Declaring an exception
6. BEGIN
7. RAISE e\_user\_error; --Raising an exception
8. EXCEPTION
9. WHEN e\_user\_error THEN --Handling an exception
10. ...
11. END;

Highlights:

 Requirement 19: HR Manager wants to view manager's details of an employee

 Develop a PL/SQL program to view the manager's details, first name and phone number of department 60. Declare, raise and handle a user-defined exception if manager is not assigned to the department.

Demosteps:

Step 1: Execute the below code in SQL Developer and observe the output.

1. DECLARE
2. v\_department\_id departments.department\_id%TYPE := 60;
3. v\_manager\_id departments.manager\_id%TYPE;
4. **e\_no\_manager EXCEPTION;**
5. v\_first\_name employees.first\_name%TYPE;
6. v\_phone\_number employees.phone\_number%TYPE;
7. BEGIN
8. --Fetching manager id of department 60 into a variable
9. SELECT manager\_id INTO v\_manager\_id FROM departments WHERE department\_id = v\_department\_id;
10. --Checking if manager is present
11. **IF v\_manager\_id IS NULL THEN**
12. **RAISE e\_no\_manager;**
13. **END IF;**
14. --Fetching manager details into variables
15. SELECT first\_name,phone\_number INTO v\_first\_name,v\_phone\_number FROM employees WHERE employee\_id=v\_manager\_id;
16. DBMS\_OUTPUT.PUT\_LINE ('First name: '||v\_first\_name);
17. DBMS\_OUTPUT.PUT\_LINE ('Phone number: '||v\_phone\_number);
18. EXCEPTION
19. --Handling of exceptions
20. **WHEN e\_no\_manager THEN**
21. **DBMS\_OUTPUT.PUT\_LINE ('Department '||v\_department\_id||' has no manager.');**
22. WHEN OTHERS THEN
23. DBMS\_OUTPUT.PUT\_LINE ('Something went wrong!!');
24. DBMS\_OUTPUT.PUT\_LINE ('Error Code: ' || SQLCODE);
25. DBMS\_OUTPUT.PUT\_LINE ('Error Message: ' || SQLERRM);
26. END;

## ProblemStatement:

# ****Requirement 20****: HR Manager wants to add a new department

Develop a PL/SQL program to add a new department with

    DEPARTMENT\_ID: 111

    DEPARTMENT\_NAME: Sales

    MANAGER\_ID: 222

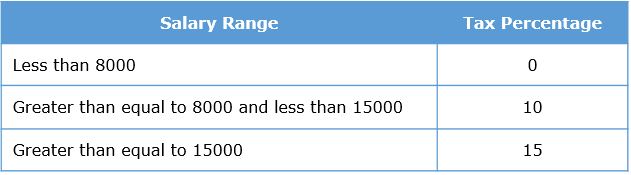
    LOCATION\_ID: 1500

In the program,

a. Declare, raise and handle a user-defined exception with an appropriate message if manager ID is not valid.

b. Declare, raise and handle a user-defined exception with an appropriate message if location ID is not valid.

**Revisiting requirement 5: An employee wants to calculate his/her tax amount based on the given salary slabs**



The below code is for calculating the tax amount for employee ID 100.

1. DECLARE
2. --used for tax calculation
3. p\_employee\_id employees.employee\_id%TYPE:=100;
4. v\_salary employees.salary%TYPE ;
5. v\_tax\_percentage NUMBER(2);
6. v\_tax\_amount NUMBER(8,2);
7. BEGIN
8. --fetching salary of the given employee
9. SELECT salary INTO v\_salary FROM employees WHERE employee\_id = p\_employee\_id;
10. --logic for tax percentage computation
11. IF v\_salary >=15000 THEN
12. v\_tax\_percentage := 15;
13. ELSIF v\_salary < 15000 AND v\_salary >= 8000 THEN
14. v\_tax\_percentage := 10;
15. ELSE
16. v\_tax\_percentage := 0;
17. END IF;
18. --formula for tax calculation
19. v\_tax\_amount:=v\_salary\*v\_tax\_percentage\*0.01;
20. DBMS\_OUTPUT.put\_line('Employee has to pay tax of Rs.'||v\_tax\_amount);
21. END;

Now, employees Shelley Higgins (employee ID 205), Matthew Weiss (employee ID 120), James Landry (employee ID 127), John Chen (employee ID 110) also want to calculate their tax.

Instead of hard coding the value of employee ID for calculation of tax with different employee IDs, you can create a **stored function**that accept employee IDs and returns the tax amount for that employee.

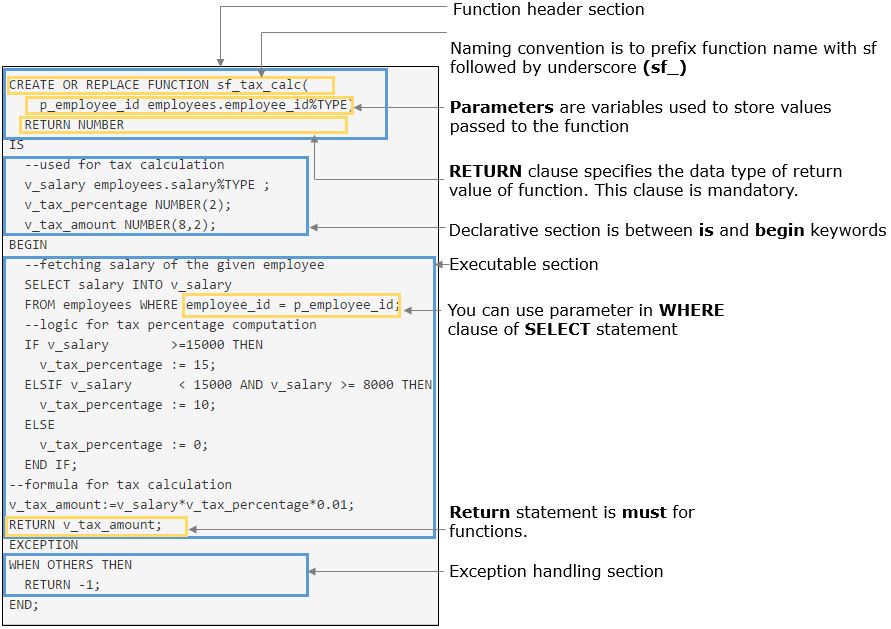
# Revisitng requirement 5: An employee wants to calculate his/her tax amount

Execute the below code to create a stored function to calculate tax of employees.

1. CREATE OR REPLACE FUNCTION sf\_tax\_calc(
2. p\_employee\_id employees.employee\_id%TYPE)
3. RETURN NUMBER
4. IS
5. --used for tax calculation
6. v\_salary employees.salary%TYPE ;
7. v\_tax\_percentage NUMBER(2);
8. v\_tax\_amount NUMBER(8,2);
9. BEGIN
10. --fetching salary of the given employee
11. SELECT salary INTO v\_salary
12. FROM employees WHERE employee\_id = p\_employee\_id;
13. --logic for tax percentage computation
14. IF v\_salary >=15000 THEN
15. v\_tax\_percentage := 15;
16. ELSIF v\_salary < 15000 AND v\_salary >= 8000 THEN
17. v\_tax\_percentage := 10;
18. ELSE
19. v\_tax\_percentage := 0;
20. END IF;
21. --formula for tax calculation
22. v\_tax\_amount:=v\_salary\*v\_tax\_percentage\*0.01;
23. RETURN v\_tax\_amount;
24. EXCEPTION
25. WHEN OTHERS THEN
26. RETURN -1;
27. END;

On creating a stored function, the function gets **compiled and stored as database object** but the program will not be executed. To execute the program you need to invoke the function.

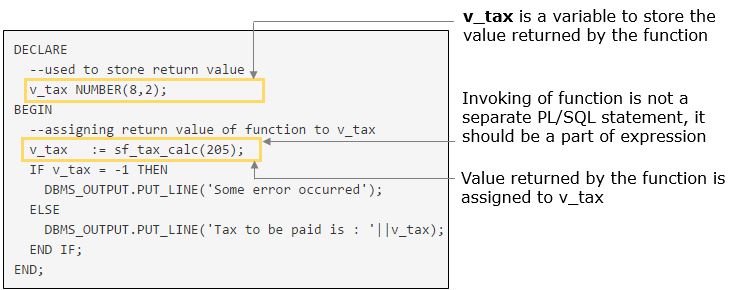
**Requirement 5: An employee wants to calculate his/her tax amount: Code anatomy**



Execute the anonymous PL/SQL block to invoke the stored function, sf\_tax\_calc.

1. DECLARE
2. --used to store return value
3. v\_tax NUMBER(8,2);
4. BEGIN
5. --assigning return value of function to v\_tax
6. v\_tax := sf\_tax\_calc(205);
7. IF v\_tax = -1 THEN
8. DBMS\_OUTPUT.PUT\_LINE('Some error occurred');
9. ELSE
10. DBMS\_OUTPUT.PUT\_LINE('Tax to be paid is : '||v\_tax);
11. END IF;
12. END;

**Invoking the stored function : Code Anatomy**



A stored function is a named PL/SQL block stored in the database that can accept parameters and **must return a single value** to the environment.

# Syntax for creating a stored function

1. CREATE [OR REPLACE ] FUNCTION function\_name [ ( parameter [ , parameter ]... ) ]
2. RETURN datatype
3. {IS | AS}
4. [ local declarations ]
5. BEGIN
6. executable statements
7. [ EXCEPTION
8. exception handlers ]
9. END [ name ];

**NOTE**:

In function declaration, it is illegal to constrain CHAR and VARCHAR2 parameters with a size, or NUMBER parameter with a precision as the constraints will be taken from actual parameters.

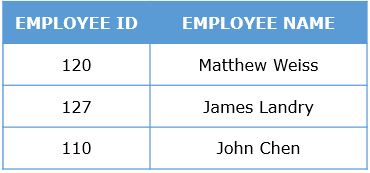
Also for RETURN clause in function declaration, to constrain CHAR and VARCHAR2 datatype with a size, or NUMBER datatype with a precision is not allowed.

The REPLACE option drops the existing stored function and replaces it with the new version.

## ProblemStatement:

# Requirement 5: An employee wants to calculate his/her tax amount

**Revisiting requirement 5**, invoke the stored function sf\_tax\_calc from anonymous block for calculating tax for below listed employees.



Also invoke the stored function sf\_tax\_calc with your employee\_id as parameter and verify the return value.

Highlights:

 Revisiting requirement 4: Employees wants to check if they are eligible to pay tax. Employees having salary greater than 8000 are eligible to pay tax.

 Develop a stored function sf\_tax\_eligibility to check if employee is eligible to pay tax.

 Invoke the function sf\_tax\_eligibility to check for tax eligibility for employee\_id 200.

Demosteps:

**Step 1**: Execute the below code in SQL Developer to create the function.

1. CREATE OR REPLACE FUNCTION sf\_tax\_eligibility(
2. p\_employee\_id employees.employee\_id%TYPE)
3. RETURN BOOLEAN
4. IS
5. --variable to store the retrieved salary
6. v\_salary employees.salary%TYPE;
7. v\_eligible\_salary CONSTANT NUMBER(6) := 8000;
8. BEGIN
9. --fetching salary of given employee id
10. SELECT salary
11. INTO v\_salary
12. FROM employees
13. WHERE employee\_id=p\_employee\_id;
14. --retrieved value is checked as per the rule
15. IF v\_salary >v\_eligible\_salary THEN
16. RETURN True;
17. ELSE
18. RETURN False;
19. END IF;
20. EXCEPTION
21. WHEN OTHERS THEN
22. RETURN NULL;
23. END;

**Step 2**: Invoking the function from anonymous PL/SQL block.

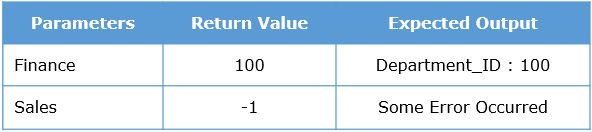
1. DECLARE
2. v\_employee\_id employees.employee\_id%TYPE :=200;
3. v\_result BOOLEAN;
4. BEGIN
5. v\_result :=sf\_tax\_eligibility(v\_employee\_id);
6. IF v\_result=True THEN
7. DBMS\_OUTPUT.PUT\_LINE('Employee is eligible to pay tax');
8. ELSIF v\_result=False THEN
9. DBMS\_OUTPUT.PUT\_LINE('Employee is not eligible to pay tax');
10. ELSE
11. DBMS\_OUTPUT.PUT\_LINE('Some error occured');
12. END IF;
13. END;

## ProblemStatement:

# ****Requirement 26****: An employee wants to view his/her department ID based on department name

Develop a stored function sf\_get\_dept\_id that accepts department name and returns department ID. In case of any error return -1.

Invoke the function from anonymous block for '**Finance**' and '**Sales**'department and display appropriate messages.



# Requirement 27: HR Manager want to calculate the tax to be paid by all the employees

If you want to display employee ID, first name, salary and tax amount of all the employees, execute the below query.

1. SELECT employee\_id,first\_name,salary,sf\_tax\_calc(employee\_id) AS tax\_amount FROM employees;

You can invoke stored function from the below clauses of SQL statements

* SELECT clause of SQL statement
* WHERE clause of a SELECT, UPDATE or DELETE statements and HAVING clauses of a SELECT statement
* VALUES clause of an INSERT statement
* SET clause of an UPDATE statement

If you want to display employee ID, first name, salary and tax amount of the employees whose tax amount is greater than 1000, execute the below query.

1. SELECT employee\_id,first\_name,salary,sf\_tax\_calc(employee\_id) AS tax\_amount
2. FROM employees WHERE sf\_tax\_calc(employee\_id)>1000;

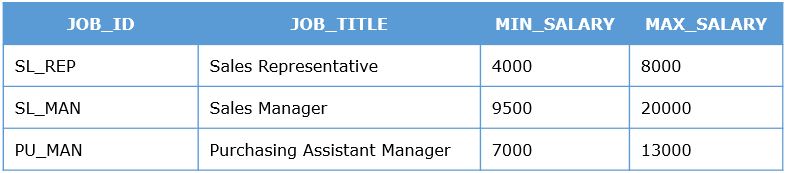
A stored function can be invoked from SQL statements only if the functions follow the below:

* The stored function should accept only IN parameters (you will learn parameter modes later in this course)
* The function should accept only SQL datatypes as parameters. The return datatypes should be only SQL datatypes
* The stored functions should not contain DML (INSERT, DELETE or UPDATE) statements
* If the stored function is invoked from an UPDATE or DELETE statement, the stored function cannot perform a SELECT on the same table
* The stored functions should not contain COMMIT or ROLLBACK

**Revisiting requirement 6: HR Manager wants to add a new job**

1. DECLARE
2. v\_job\_ID VARCHAR2(10) := 'FI\_CLERK';
3. v\_job\_title VARCHAR2(35) := 'Finance Clerk';
4. v\_min\_salary NUMBER(6) := 5000;
5. v\_max\_salary NUMBER(6) := 10000;
6. v\_count\_of\_jobs pls\_integer;
7. BEGIN
8. SELECT COUNT(Job\_id) INTO v\_count\_of\_jobs FROM jobs WHERE job\_id=v\_job\_ID;
9. IF(v\_count\_of\_jobs=0) THEN
10. INSERT INTO Jobs VALUES(v\_job\_ID,v\_job\_title,v\_min\_salary,v\_max\_salary);
11. ELSE
12. DBMS\_OUTPUT.PUT\_LINE(v\_job\_title||' already exists');
13. END IF;
14. END;

Now HR Manager wants to add 3 new jobs after checking if the same job ID does not exist in the table.



Instead of modifying and executing the code with different hard coded values, you can have a **stored procedure sp\_add\_job**  with **p\_job\_id**, **p\_job\_title**, **p\_min\_salary**, **p\_max\_salary, p\_status**as parameters to carry out this action and invoke it with different values of parameters.

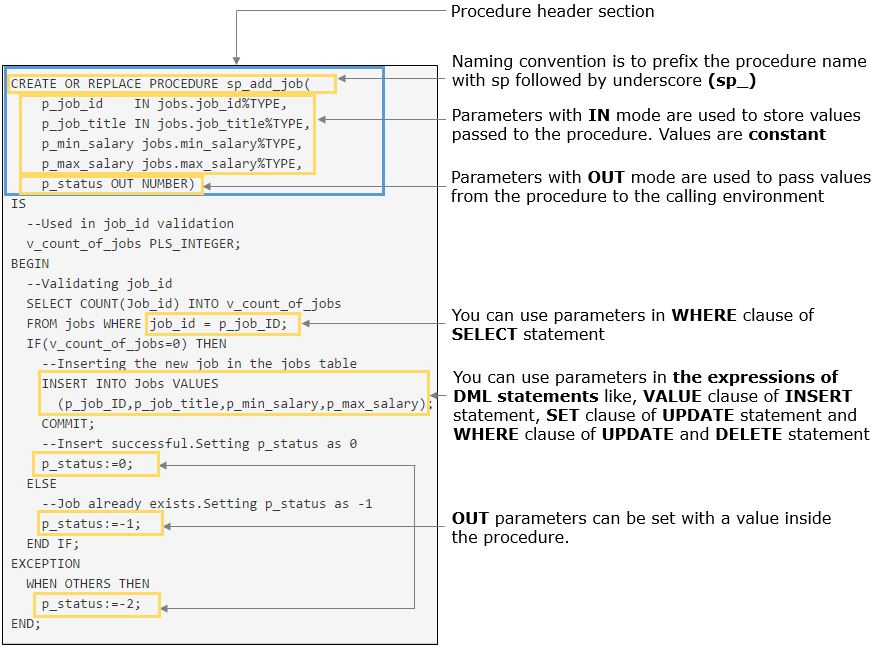
# Revisitng requirement 6: HR Manager wants to add a new job

Execute the below code to create the stored procedure to add a job.

1. CREATE OR REPLACE PROCEDURE sp\_add\_job(
2. p\_job\_id IN jobs.job\_id%TYPE,
3. p\_job\_title IN jobs.job\_title%TYPE,
4. p\_min\_salary jobs.min\_salary%TYPE,
5. p\_max\_salary jobs.max\_salary%TYPE,
6. p\_status OUT NUMBER)
7. IS
8. --Used in job\_id validation
9. v\_count\_of\_jobs PLS\_INTEGER;
10. BEGIN
11. --Validating job\_id
12. SELECT COUNT(Job\_id) INTO v\_count\_of\_jobs
13. FROM jobs WHERE job\_id = p\_job\_ID;
14. IF(v\_count\_of\_jobs=0) THEN
15. --Inserting the new job in the jobs table
16. INSERT INTO Jobs VALUES
17. (p\_job\_ID,p\_job\_title,p\_min\_salary,p\_max\_salary);
18. COMMIT;
19. --Insert successful.Setting p\_status as 0
20. p\_status:=0;
21. ELSE
22. --Job already exists.Setting p\_status as -1
23. p\_status:=-1;
24. END IF;
25. EXCEPTION
26. WHEN OTHERS THEN
27. p\_status:=-2;
28. END;

Similar to stored functions, the **stored procedure** also gets **compiled and stored as database object**. To execute the stored procedure, you need to invoke the procedure explicitly.

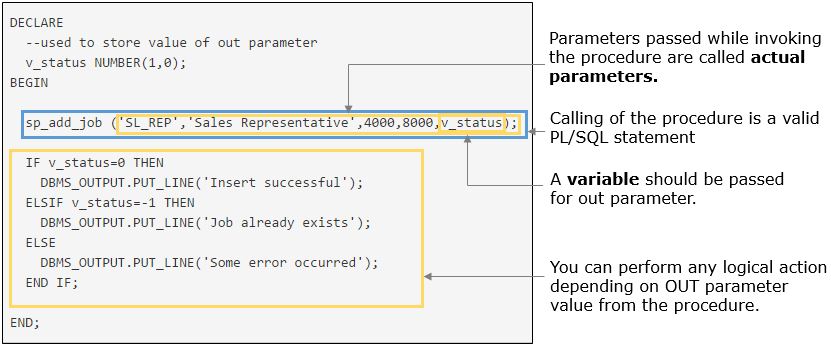
**Requirement 6: HR Manager wants to add a new job: Code anatomy**



Execute the below code to invoke the procedure sp\_add\_job and observe the output.

1. DECLARE
2. --used to store value of out parameter
3. v\_status NUMBER(1,0);
4. BEGIN
5. sp\_add\_job ('SL\_REP','Sales Representative',4000,8000,v\_status);
6. IF v\_status=0 THEN
7. DBMS\_OUTPUT.PUT\_LINE('Insert successful');
8. ELSIF v\_status=-1 THEN
9. DBMS\_OUTPUT.PUT\_LINE('Job already exists');
10. ELSE
11. DBMS\_OUTPUT.PUT\_LINE('Some error occurred');
12. END IF;
13. END;

**Invoking stored procedure: Code Anatomy**



A stored procedure is a named PL/SQL block that can accept parameters and be invoked. You use a procedure to perform an action.

1. CREATE [OR REPLACE ] PROCEDURE procedure\_name [ ( parameter [ , parameter ]... ) ]
2. {IS | AS}
3. [ local declarations ]
4. BEGIN
5. executable statements
6. [ EXCEPTION
7. exception handlers ]
8. END [ name ];

**NOTE:**

In procedure declaration, it is illegal to constrain CHAR and VARCHAR2 parameters with a size, or NUMBER parameter with a precision, as the constraints will be taken from actual parameters.

The REPLACE option drops the existing stored procedure and replaces it with the new version.

A procedure cannot return any value directly through the RETURN statement. However, a procedure can contain a RETURN statement (Example: RETURN;) to designate the end of the procedure.

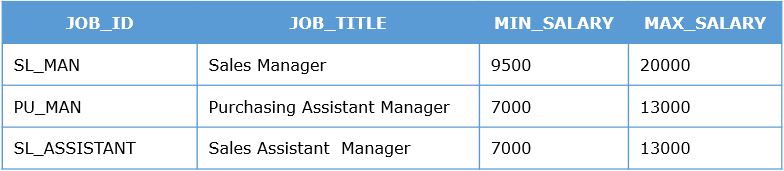
## ProblemStatement:

# ****Revisiting requirement 6****: HR Manager wants to add a new job

HR manager wants to add below listed new jobs after checking if the same job ID does not exist in the table.

Use procedure **sp\_add\_job**for inserting new jobs and getting the status of insertion. If OUT parameter value is 0, display '**New job is added successfully**', if  -1, display '**Job\_id already exist**', else display '**Some error occurred**'.

After invoking the procedure, verify the result in Jobs table.



Highlights:

* Requirement 28: HR Manager wants to add a new region in regions table.
* Develop a stored procedure sp\_add\_region which accepts region\_id and region\_name as IN parameter and set the OUT parameter status as 0 for successful insertion, -1 if region\_id already exists and -2 if region\_name already exists. In case of exception, set the status as -3.
* Invoke the procedure with parameters as (6, Australia).

Demosteps:

**Step 1**: Execute the below code to create procedure sp\_add\_region.

1. CREATE OR REPLACE PROCEDURE sp\_add\_region(
2. p\_region\_id regions.region\_id%TYPE,
3. p\_region\_name regions.region\_name%TYPE,
4. p\_status OUT NUMBER)
5. IS
6. --used for validation of region\_id and region\_name
7. v\_count\_of\_id NUMBER(3);
8. v\_count\_of\_name NUMBER(3);
9. BEGIN
10. --checking if given region\_id already exist
11. SELECT COUNT(region\_id) INTO v\_count\_of\_id FROM regions WHERE region\_id=p\_region\_id;
12. IF v\_count\_of\_id=0 THEN
13. --checking if given region\_id already exist
14. SELECT COUNT(region\_name) INTO v\_count\_of\_name FROM regions WHERE region\_name=p\_region\_name;
15. IF v\_count\_of\_name=0 THEN
16. --Inserting new region details afte successful validation
17. INSERT INTO regions VALUES(p\_region\_id,p\_region\_name);
18. COMMIT;
19. --Successful insert
20. p\_status:=0;
21. ELSE
22. --Region\_name already exist
23. p\_status:=-2;
24. RETURN; --procedure execution will terminate if region\_name already exist
25. END IF;
26. ELSE
27. --Region\_id already exist
28. p\_status:=-1;
29. RETURN; --procedure execution will terminate if region\_id already exist
30. END IF;
31. EXCEPTION
32. WHEN OTHERS THEN
33. p\_status:=-3;
34. END;

**Step 2**:Execute the below code to invoke the procedure from anonymous block.

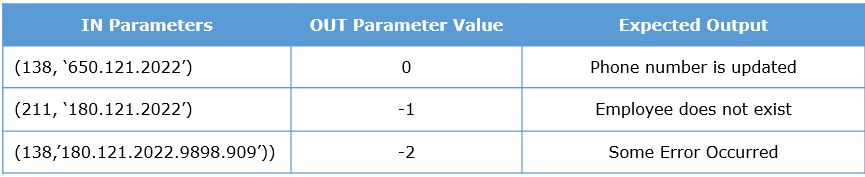
1. DECLARE
2. v\_region\_id regions.region\_id%TYPE := 6;
3. v\_region\_name regions.region\_name%TYPE := 'Australia';
4. v\_status NUMBER(1);
5. BEGIN
6. sp\_add\_region(v\_region\_id,v\_region\_name,v\_status);
7. IF v\_status=0 THEN
8. DBMS\_OUTPUT.PUT\_LINE('Insert successful');
9. ELSIF v\_status=-1 THEN
10. DBMS\_OUTPUT.PUT\_LINE('Region id already exist');
11. ELSIF v\_status=-2 THEN
12. DBMS\_OUTPUT.PUT\_LINE('Region name already exist');
13. ELSE
14. DBMS\_OUTPUT.PUT\_LINE('Some error occurred');
15. END IF;
16. END;

## ProblemStatement:

# ****Revisiting requirement 7:****An employee wants to update his/her phone number

Develop a stored procedure sp\_update\_contact which accepts employee ID and new phone number as IN parameters, status as OUT parameter, status is set as 0 if update is successful, -1 if employee\_id is invalid and -2 in case of any other exception.

Invoke the created procedure with following values.

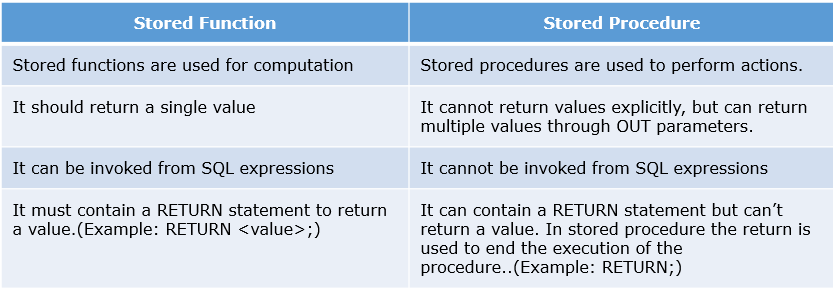


Subprogram is a **named PL/SQL block**that can be stored in the database, hence can be **reused**easily. **Stored function and stored procedure** are the types of subprogram.

Subprograms are compiled once when they are created and can be executed any number of times by invoking it from any PL/SQL block.

As you have seen, stored **functions**are used to **compute values**and stored **procedures**can be used for**performing actions**.

**Comparison of stored function and stored procedure**



For more detailed information on subprograms, you can refer [Subprogram](https://docs.oracle.com/cd/B28359_01/appdev.111/b28370/subprograms.htm)

ou can remove stored functions and stored procedure from the database.

# Syntax for removing a stored function

1. DROP FUNCTION <function\_name>;

Execute the below code to remove the stored function sf\_tax\_calc from the database.

1. DROP FUNCTION sf\_tax\_calc;

# Syntax for removing a stored procedure

1. DROP PROCEDURE <procedure\_name>;

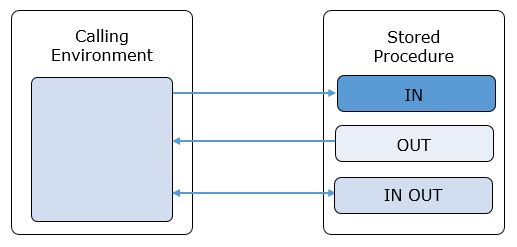
Execute the below code to remove the stored procedure sp\_add\_job from the database.

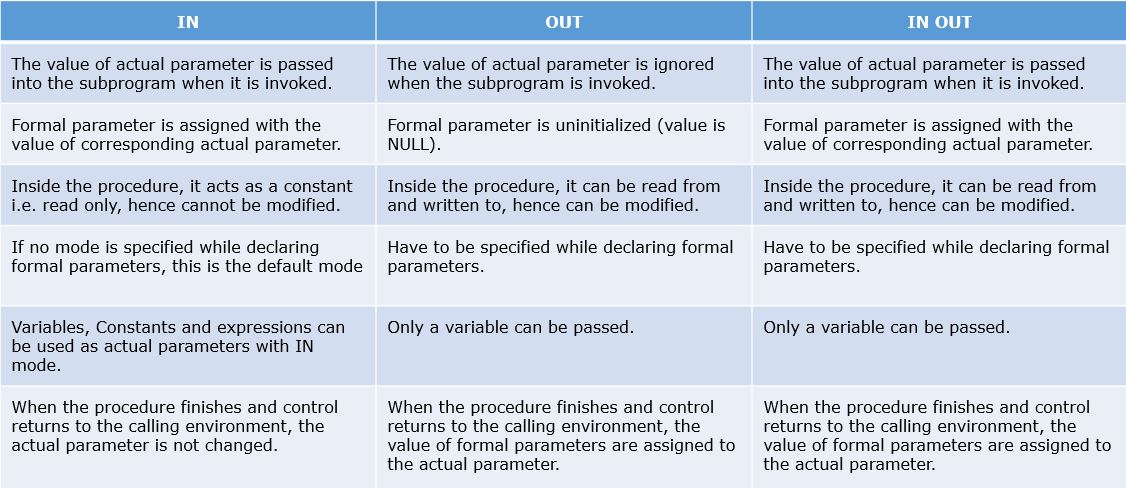
1. DROP PROCEDURE sp\_add\_job;

Subprogram parameters are used to receive data from, and send results to, a calling program.

There are 3 parameter modes that determines the direction of data transfer:

* IN
* OUT
* IN OUT





**DEFAULT**value can be assigned only to IN parameter that are used when actual parameter values are not passed.

Execute the below code to create a stored function sp\_bonus\_amount that calculates the bonus amount of an employee. The function accepts employee ID and bonus percentage and returns the bonus amount.

1. CREATE OR REPLACE FUNCTION sp\_bonus\_amount(
2. p\_employee\_id employees.employee\_id%TYPE,
3. **p\_bonus\_percentage NUMBER DEFAULT 10**)
4. RETURN NUMBER
5. IS
6. v\_bonus\_amount NUMBER(8,2);
7. v\_salary employees.salary%TYPE;
8. BEGIN
9. SELECT salary INTO v\_salary FROM employees WHERE employee\_id=p\_employee\_id;
10. v\_bonus\_amount:=v\_salary\*(p\_bonus\_percentage/100);
11. RETURN v\_bonus\_amount;
12. END;

Invoke the function for employee ID 100 without giving bonus percentage.

1. DECLARE
2. v\_employee\_id employees.employee\_id%TYPE:=100;
3. v\_bonus\_amount NUMBER(8,2);
4. BEGIN
5. v\_bonus\_amount:=**sp\_bonus\_amount(v\_employee\_id)**;
6. DBMS\_OUTPUT.PUT\_LINE('Bonus amount is '|| v\_bonus\_amount);
7. END;

# Observation:

Since bonus percentage is not passed to the function, it will take bonus percentage as the DEFAULT value i.e. 10.

Execute the below code to invoke sp\_bonus\_amount with employee ID 100 and bonus percentage as 50.

1. DECLARE
2. v\_employee\_id employees.employee\_id%TYPE:=100;
3. v\_bonus\_amount NUMBER(8,2);
4. v\_bonus\_percentage NUMBER(5,2):=50;
5. BEGIN
6. v\_bonus\_amount:=sp\_bonus\_amount(v\_employee\_id,v\_bonus\_percentage);
7. DBMS\_OUTPUT.PUT\_LINE('Bonus amount is '|| v\_bonus\_amount);
8. END;

# Observation:

DEFAULT value of p\_bonus\_percentage(=10) is overridden in this instance for computing the bonus amount.

PL/SQL gives you 3 ways to pass actual parameters. Consider the stored procedure sp\_add\_region

1. CREATE OR REPLACE PROCEDURE sp\_add\_region(
2. p\_region\_id regions.region\_id%TYPE,
3. p\_region\_name regions.region\_name%TYPE,
4. p\_status OUT NUMBER)

* **Positional notation**:The actual parameters are associated with the formal parameters by position

1. sp\_add\_region(6,'Australia',v\_status);

* **Named notation**: The formal parameters and the actual parameter are both included for each parameter. The order of parameters can be rearranged, if desired

1. sp\_add\_region(p\_region\_name=>'Australia',p\_status=>v\_status,p\_region\_id=>6);

* **Mixed notation**:The first parameters should be specified using positional notation and the remaining parameters can be specified using named notation. The reverse is not allowed

1. sp\_add\_region(6,p\_status=>v\_status,p\_region\_name=>'Australia');

Triggers:

# ****Requirement 40****: HR Manager wants to keep track of salary changes of every employee

HR Manager wants to keep track of salary changes of every employee for auditing in the future. Whenever an HR Manager modifies the salary of any employee, the below details need to be inserted in **EMP\_SALARY\_LOG** table

   employee\_id: employee ID of whose salary is getting modified

   modified\_time: at what time this modification takes place

   old\_salary: existing salary of employee

   new\_salary: modified salary

   user\_name: database user who is making this modification

# Execute the below script in SQL Developer to create ****EMP\_SALARY\_LOG**** table:

1. CREATE TABLE emp\_salary\_log
2. (
3. employee\_id   NUMBER(6),
4. modified\_time TIMESTAMP,
5. old\_salary    NUMBER(6),
6. new\_salary    NUMBER(6),
7. user\_name     VARCHAR2(30)
8. )

# Execute the below PL/SQL code in SQL Developer:

1. CREATE OR REPLACE TRIGGER trg\_emp\_sal\_log
2. BEFORE  UPDATE OF salary ON employees
3. FOR EACH ROW
4. DECLARE
5. -- old and new are pseudo records to retreive previous and modified values of the column
6. v\_old\_sal NUMBER := :old.salary;
7. v\_new\_sal   NUMBER := :new.salary;
8. BEGIN
9. INSERT INTO emp\_salary\_log VALUES (:new.employee\_id,systimestamp, v\_old\_sal,v\_new\_sal,USER);
10. -- USER will retreive the current login database user name
11. END;

**Note:** When you execute the above code, a Trigger will be compiled and stored as a **database object**similar to subprograms. However, you cannot explicitly invoke this trigger in PL/SQL.

HR manager wants to give salary hike of 10% to employees with Job ID 'AD\_ASST'

# Execute the below SQL Statement in SQL Developer and observe the values of salary:

1. SELECT EMPLOYEE\_ID,SALARY,JOB\_ID FROM employees WHERE JOB\_ID = 'AD\_ASST';

# Execute the below Update Statement in SQL Developer:

1. UPDATE EMPLOYEES SET SALARY = SALARY\*1.1 WHERE JOB\_ID = 'AD\_ASST';

**Note:** When you execute the update statement, the trigger will get executed and salary details will be entered into **EMP\_SALARY\_LOG** table

# Execute the below SQL Statements in SQL Developer and observe:

1. SELECT EMPLOYEE\_ID,SALARY,JOB\_ID FROM employees WHERE JOB\_ID = 'AD\_ASST';
2. SELECT EMPLOYEE\_ID,MODIFIED\_TIME,OLD\_SALARY,NEW\_SALARY,USER\_NAME FROM emp\_salary\_log;

Execute the same update statement for Job ID 'IT\_PROG' and observe the number of rows inserted in EMP\_SALARY\_LOG

**Note:** For every employee whose salary is changed, a new row is inserted in **EMP\_SALARY\_LOG** table.

The code you just executed is a Trigger.

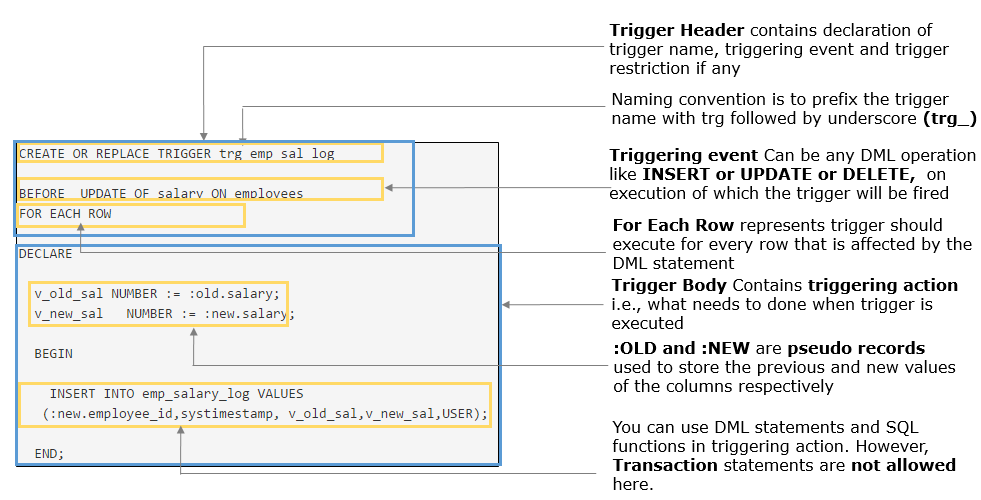
A **Trigger** is a stored PL/SQL program unit, which gets implicitly fired by Oracle Database when a triggering event occurs. A Trigger defines an action, which should get executed, when some database related event occurs.

There are [different kind of triggers](https://docs.oracle.com/cd/B28359_01/server.111/b28318/triggers.htm#i13313). In this course you will be using only DML Triggers.

DML Triggers are associated with a specific database table. The purpose of this trigger is to perform a specific service when a specified DML operation (INSERT, UPDATE, DELETE) occurs on a table.

Unlike the stored procedures (or functions) which have to be explicitly invoked, these triggers implicitly gets fired (executed) whenever the table is affected by any DML operation.

**Requirement 40: HR Manager wants to keep track of salary changes of every employee: Code anatomy**



A  trigger has three basic parts:

* **Triggering event (or Statement)** : An SQL statement which causes a trigger to be fired. Every triggering event is associated with trigger timing i.e., when a trigger should execute. According to trigger timing, triggers can be either **before or after** triggers. Triggering events can be
  + Before INSERT or After INSERT
  + Before UPDATE or After UPDATE
  + Before DELETE or After DELETE
* **Trigger constraint (Optional)**: A Boolean expression which must be TRUE for the trigger to be fired. We can add triggering constraint by using a WHEN clause in trigger.
* **Trigger action**: Is a PL/SQL code which contains what needs to be done when a trigger is fired.

# Structure of Trigger:

1. CREATE [OR REPLACE] TRIGGER <trigger-name> BEFORE | AFTER | INSTEAD OF
2. DELETE | [OR] INSERT | [OR] UPDATE [ OF <column> [, <column>...]] ON <table>
4. -- This section is called Triggering event
6. [ FOR EACH ROW [ WHEN <condition>] ]
8. -- When clause will be triggering constraint
9. BEGIN
11. -- This PL/SQL block is Triggering action
13. */\* PL/SQL Block \*/*
14. ...
15. END;

You have one more kind of trigger called [Instead of Triggers.](https://docs.oracle.com/cd/B28359_01/server.111/b28318/triggers.htm#i10900) You won't be using this trigger in this course.

In the previous requirement you have observed that for every employee whose salary is modified, trigger is fired and a new row is inserted in EMP\_SALARY\_LOG table. These kinds of triggers are called Row Triggers.

In a**Row Trigger**, the trigger body is executed depending on how many rows were affected by the DML statement. If the DML statement did not affect any rows, the trigger body executes ZERO times.

These row triggers will have two **pseudo records** to get data from triggering event (INSERT, UPDATE, DELETE).

* **OLD:** references the OLD row before UPDATE or DELETE
* **NEW:** references the NEW row after INSERT or UPDATE

Row triggers will be used if trigger action depends on the data provided by triggering statement.

# Requirement 41: Employee salary should be greater than or equal to minimum salary of his job

Every employee salary should be greater than or equal to the minimum salary of his job. Before adding any employee this requirement has to be checked. If the salary is less than the minimum salary, then employee's salary should be inserted with the minimum salary of his job.

# Execute the below PL/SQL code in SQL Developer:

1. CREATE OR REPLACE TRIGGER trg\_check\_sal
2. **BEFORE INSERT ON EMPLOYEES**
3. **FOR EACH ROW**
4. -- Before inserting every employee this trigger will get executed
5. DECLARE
6. v\_min\_salary   jobs.min\_salary%TYPE;
7. v\_new\_job\_id   jobs.job\_id%TYPE := :new.job\_id;
8. -- To access new value of a row we need to use :new.columnname
9. BEGIN
10. -- Logic to retreive minimuM salary of a given job
11. SELECT min\_salary  INTO v\_min\_salary FROM jobs WHERE job\_id = UPPER (v\_new\_job\_id);
13. */\*Logic to check if new salary is valid or not,*
14. *if not then salary of employee should be inserted with minimum salary of his job*
15. *\*/*
16. IF :new.salary < v\_min\_salary   THEN
17. :new.salary := v\_min\_salary;
18. END IF;
19. END;

# 

HR Manager wants to add a new employee with the following details:

    Employee\_id: 1001

    First\_name: Jack

    Last\_name: Walney

    Email:  JWalney

    Hire\_date: 23-JAN-2015

    salary: 1500

   Job\_id: IT\_PROG

   department\_id: 90

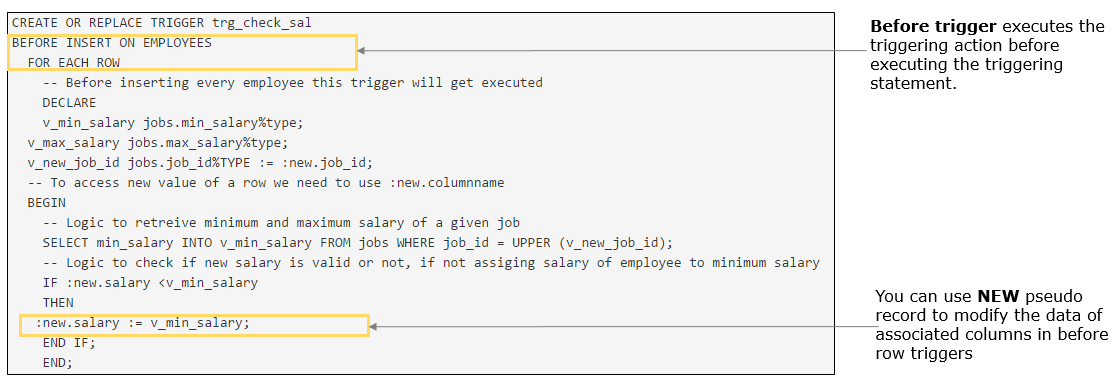
# Execute the below SQL statement in SQL Developer and observe minimum salary of Job IT\_PROG:

1. SELECT min\_salary FROM jobs WHERE job\_id = 'IT\_PROG';

# Execute the below Insert statement in SQL Developer:

1. INSERT INTO EMPLOYEES(EMPLOYEE\_ID,FIRST\_NAME,LAST\_NAME,EMAIL,HIRE\_DATE,SALARY,JOB\_ID,DEPARTMENT\_ID)
2. VALUES
3. (1001,'Jack','Walney','JWalney','23-JAN-2015',1500,'IT\_PROG',90);

# Execute the below SQL statement in SQL Developer and Observe:

1. SELECT salary FROM employees WHERE employee\_id =1001;
2. The code you just executed is a before row trigger.
3. **Before Trigger** executes trigger action (PL/SQL code) before executing triggering statement (INSERT,  UPDATE, DELETE).
4. **Requirement41: Employee salary should be greater than or equal to the minimum salary of his job: Code anatomy**
6. 

Highlights:

 Revisiting requirement 41: Whenever employee salary is modified, updated salary should be greater than or equal to the minimum salary of his job. If the salary is less than the minimum salary, then employee's salary should be inserted with the minimum salary of his job.

 Develop a trigger to check this before modifying salary of any employee

Demosteps:

**Note:** You can achieve this requirement in two methods.

**Method 1:**Creating a new trigger

**Step 1:** Execute the below PL/SQL code in SQL Developer

1. CREATE OR REPLACE TRIGGER trg\_check\_sal\_before\_update
2. BEFORE UPDATE OF SALARY
3. ON EMPLOYEES
4. FOR EACH ROW
5. -- Before updating every employee this trigger will get executed
6. DECLARE
7. v\_min\_salary   jobs.min\_salary%TYPE;
8. v\_new\_job\_id   jobs.job\_id%TYPE := :new.job\_id;
9. -- To access new value of a row we need to use :new.columnname
10. BEGIN
11. -- Logic to retreive minimum salary of a given job
12. SELECT min\_salary INTO v\_min\_salary FROM jobs WHERE job\_id = UPPER (v\_new\_job\_id);
14. -- Logic to check if new salary is valid or not, if not employees salary should be inserted with minimum salary
15. IF :new.salary < v\_min\_salary
16. THEN
17. :new.salary := v\_min\_salary;
18. END IF;
19. END;

**Test Cases:**

* Update a row with valid salary and observe
* Update a row with salary less than the minimum salary of his job and observe

Highlights:

 Revisiting requirement 41: Whenever employee salary is modified, updated salary should be greater than or equal to the minimum salary of his job. If the salary is less than the minimum salary, then employee's salary should be inserted with the minimum salary of his job.

 Develop a trigger to check this before modifying salary of any employee

Demosteps:

In the previous demo you have created a new trigger.

However, if the column of a table (**salary of employees**) that is getting affected is same for more than one triggering events (**INSERT, UPDATE, DELETE**), then you can use only one trigger to achieve both requirements.

**Method 2:**Modify the existing trigger to accommodate the new requirement

**Step 1:** Execute the below PL/SQL code in SQL Developer

1. CREATE OR REPLACE TRIGGER trg\_check\_sal
2. BEFORE INSERT OR UPDATE OF SALARY ON EMPLOYEES
3. FOR EACH ROW
4. DECLARE
5. v\_min\_salary jobs.min\_salary%type;
6. v\_new\_job\_id jobs.job\_id%TYPE := :new.job\_id;
7. -- To access new value of a row we need to use :new.columnname
8. BEGIN
9. -- Logic to retreive minimum and maximum salary of a given job
10. SELECT min\_salary INTO v\_min\_salary FROM jobs WHERE job\_id = UPPER (v\_new\_job\_id);
11. -- Logic to check if new salary is valid or not, if not assiging salary of employee to minimum salary
12. IF :new.salary <v\_min\_salary
13. THEN
14. :new.salary := v\_min\_salary;
15. END IF;
16. END;

**Test Cases:**

* Update a row with valid salary and observe
* Update a row with salary less than the minimum salary of his job and observe

# Requirement 45: HR manager cannot modify the salary of employees in the month of March

Financial year of company is from APRIL to next MARCH. At the end of financial year all balances has to be calculated. So, during this time no modification of salary should be allowed.

# Execute the below PL/SQL code in SQL Developer:

1. CREATE OR REPLACE TRIGGER trg\_check\_update\_sal
2. BEFORE UPDATE OF salary   ON EMPLOYEES
3. DECLARE
4. v\_todays\_DATE   DATE;
5. BEGIN
6. v\_todays\_Date := SYSDATE;
8. --different values for v\_todays\_date for testing the trigger execution
9. --v\_todays\_Date := '01-MAR-2016';
10. --v\_todays\_date := '31-Mar-2016';
11. --v\_todays\_date := '01-Apr-2016';
13. --logic to check if month is march or not
14. IF TO\_CHAR (v\_todays\_Date, 'MON') = 'MAR'
15. THEN
16. RAISE\_APPLICATION\_ERROR (-20001,
17. 'Employee salary cannot be modified in March');
18. END IF;
19. END;

HR Manager wants to give hike of 10% to employees working in Job 'IT\_PROG'

# Execute the below  SQL statement in SQL Developer and observe:

1. UPDATE EMPLOYEES SET SALARY = SALARY\*1.1 WHERE JOB\_ID = 'IT\_PROG';

**Note:** Modify the trigger with different values for v\_todays\_date. Execute the update statement again and observe.

The PL/SQL code you just executed is a statement trigger.

In a **statement trigger**, the trigger body is executed only once for the statement, irrespective of how many number of rows are affected by the DML statement.

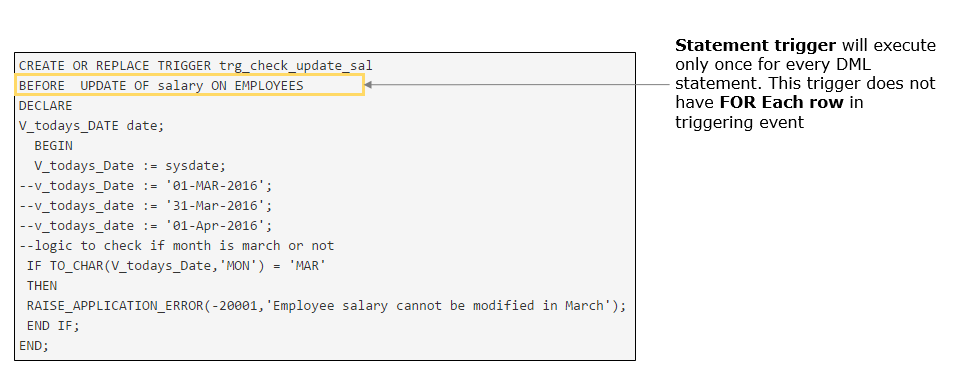
E.g: If an UPDATE statement affects 10 rows or 1 row or  0 rows, the trigger body executes once.

FOR EACH ROW clause decides whether it is a row level or statement trigger.

Statement triggers are useful in complex security check.

Unlike row triggers, there are **no pseudo records** associated with statement triggers

**Requirement 44: HR manager cannot modify the salary of employees in the month of March: Code Anatomy**



# Execute the below SQL statement in SQL Developer to view the details of trigger which is already created:

1. SELECT trigger\_name, trigger\_type, triggering\_event,
2. table\_name, referencing\_names,
3. status, trigger\_body
4. FROM user\_triggers
5. WHERE trigger\_name = <triggername>;

# Execute the below SQL statement in SQL Developer to drop a trigger:

1. DROP TRIGGER <triggername>;

When a table is dropped, all triggers associated with the trigger will be dropped automatically

**Note:** You should have necessary privileges to execute these queries. Modify <triggername> with the name of the trigger you want

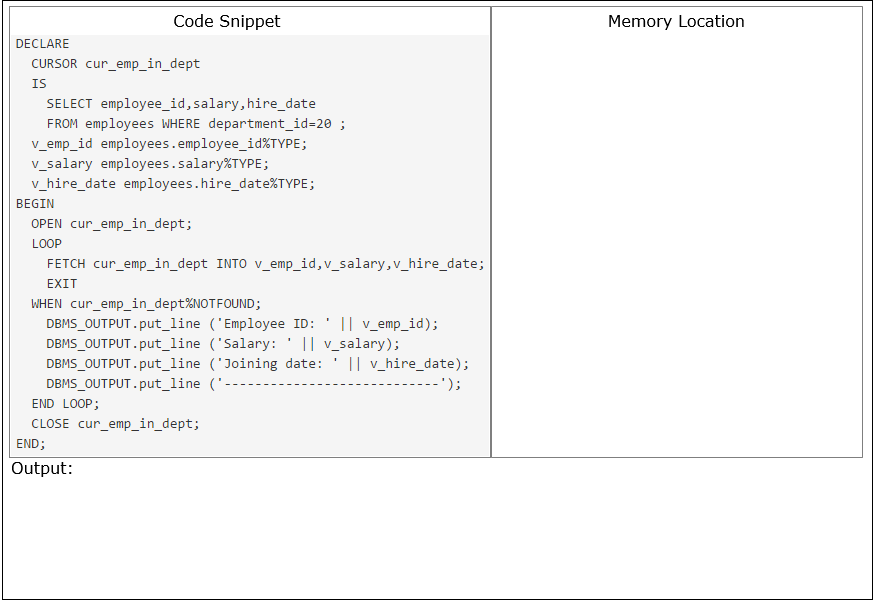
Cursors:

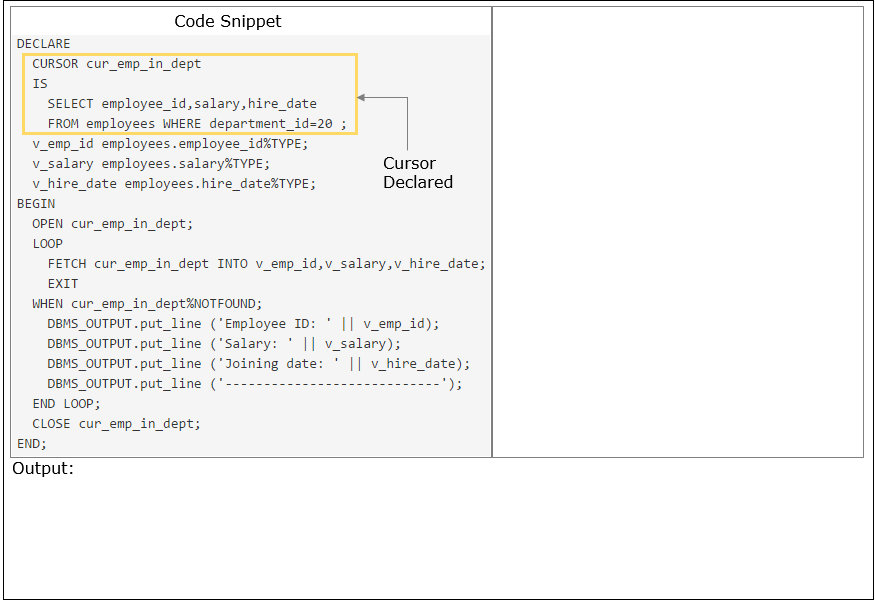
# ****Requirement 47:****HR manager wants to view the details (employee ID, salary and hire date) of all the employees working in Marketing department (department ID 20)

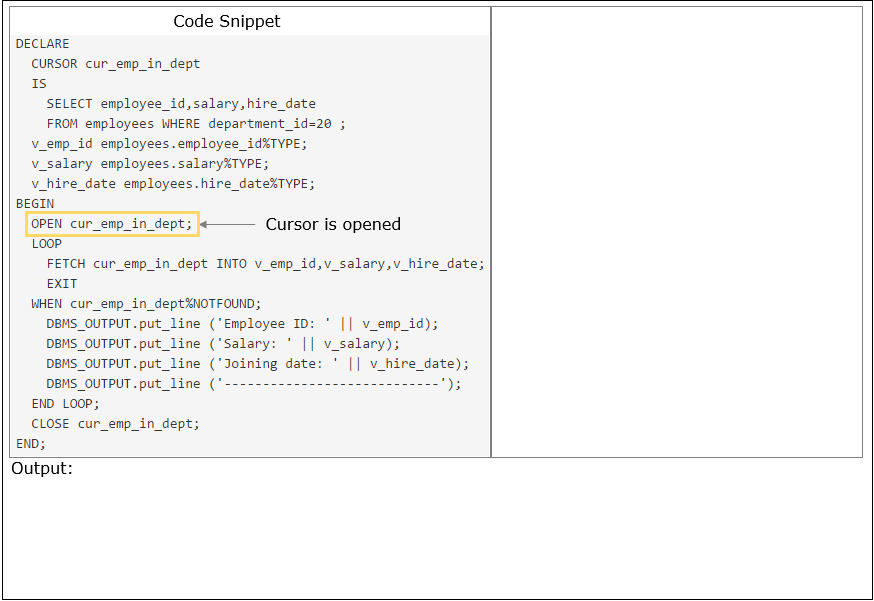
So far, you have developed programs that retrieve and processes only one row value using a SELECT INTO statement. **You can use cursor for multi-row processing**.

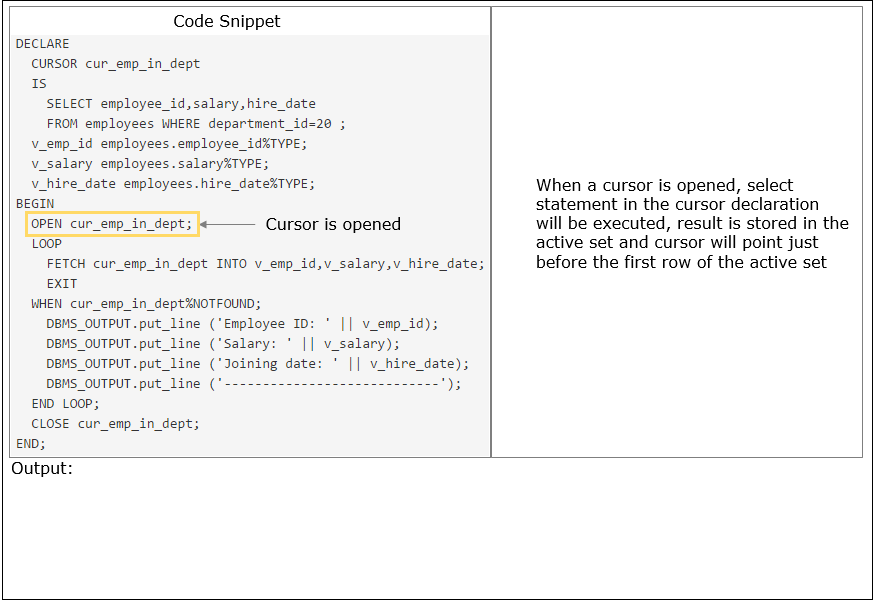
# Execute the below existing program in SQL developer and observe:

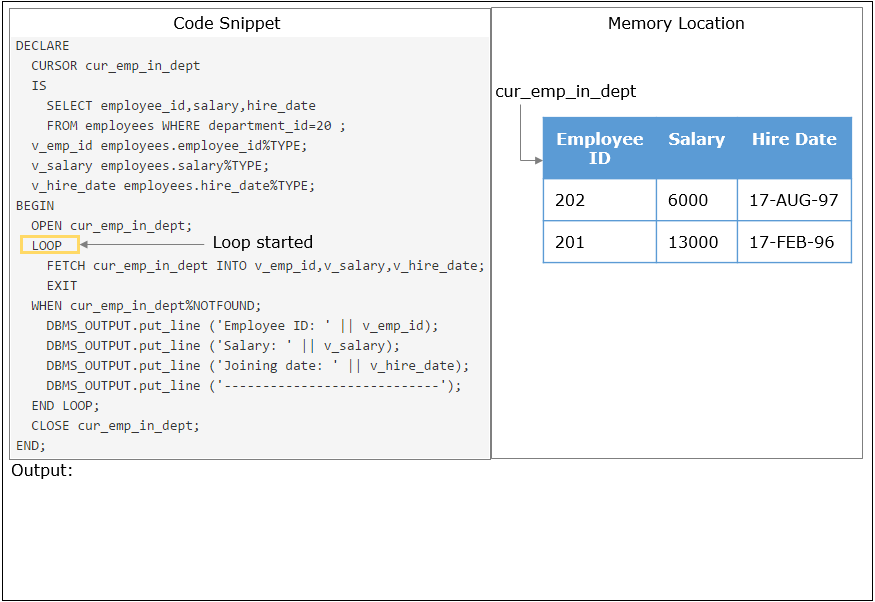
1. DECLARE
2. --declaring a cursor
3. CURSOR cur\_emp\_in\_dept
4. IS
5. SELECT employee\_id,salary,hire\_date FROM employees WHERE department\_id=20 ;
6. --declaring variables to retrieve the value
7. v\_emp\_id employees.employee\_id%TYPE;
8. v\_salary employees.salary%TYPE;
9. v\_hire\_date employees.hire\_date%TYPE;
10. BEGIN
11. --cursor is opened
12. OPEN cur\_emp\_in\_dept;
13. --loop for fetching multiple rows
14. LOOP
15. --retrieving the values using FETCH statement into the variables
16. FETCH cur\_emp\_in\_dept INTO v\_emp\_id,v\_salary,v\_hire\_date;
17. --Exit condition for loop
18. EXIT
19. WHEN cur\_emp\_in\_dept%NOTFOUND;
20. DBMS\_OUTPUT.put\_line ('Employee ID: ' || v\_emp\_id);
21. DBMS\_OUTPUT.put\_line ('Salary: ' || v\_salary);
22. DBMS\_OUTPUT.put\_line ('Joining date: ' || v\_hire\_date);
23. DBMS\_OUTPUT.put\_line ('----------------------------');
24. END LOOP;
25. --cursor is closed
26. CLOSE cur\_emp\_in\_dept;
27. END;

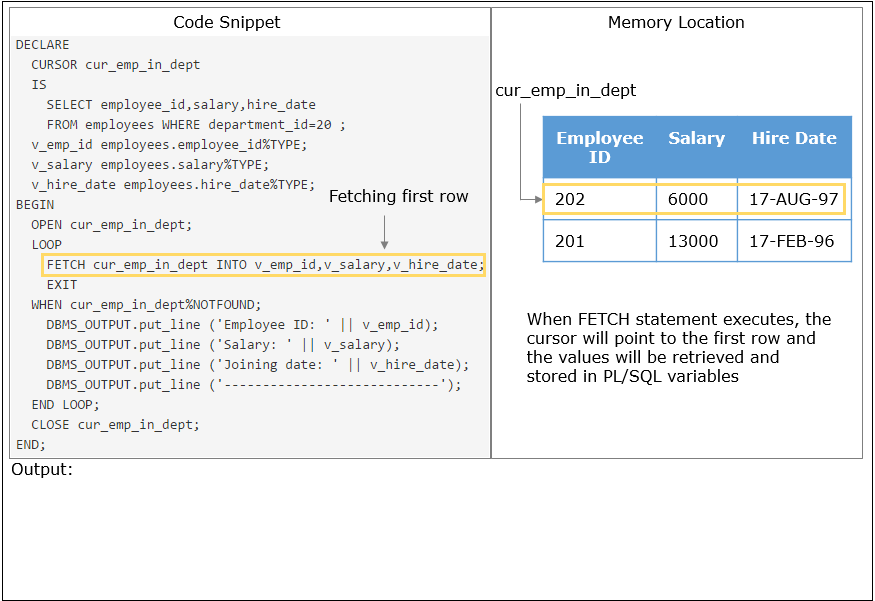
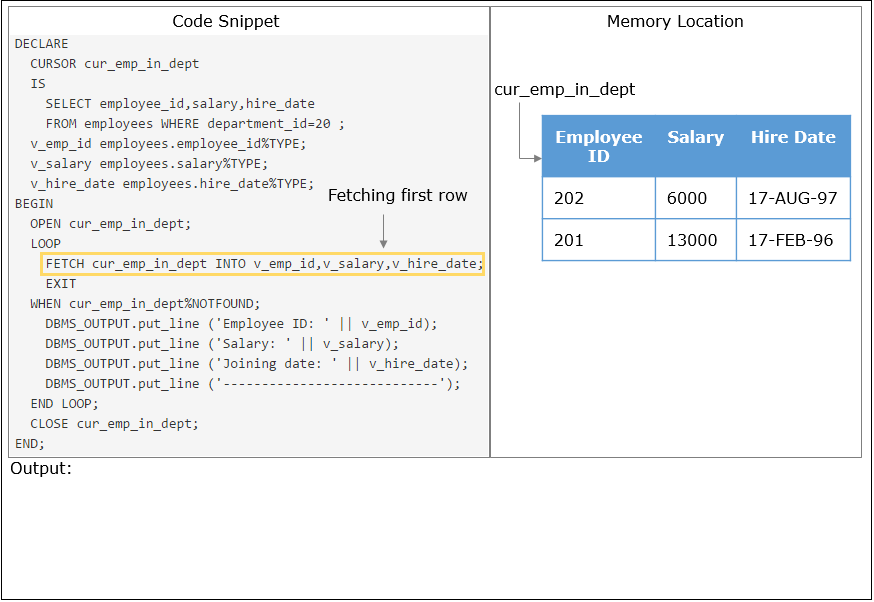


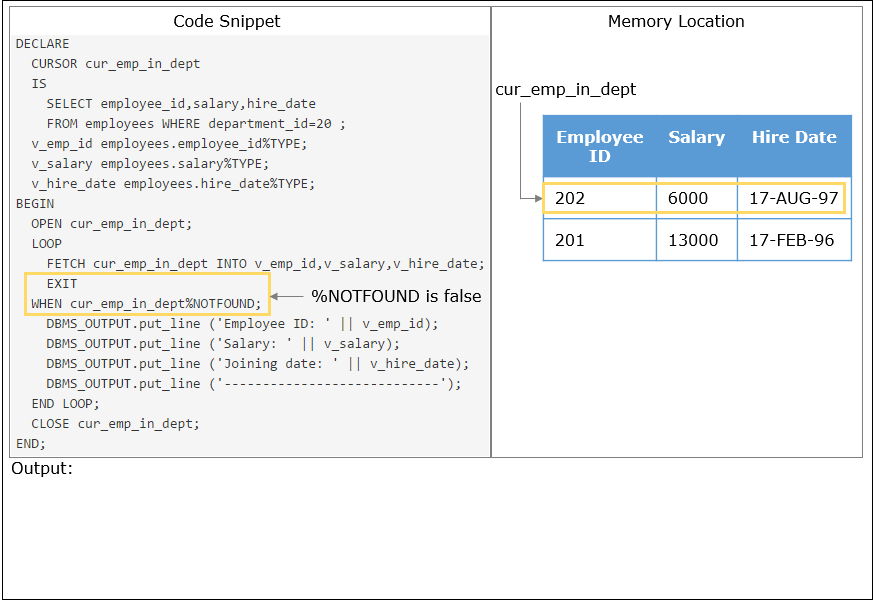


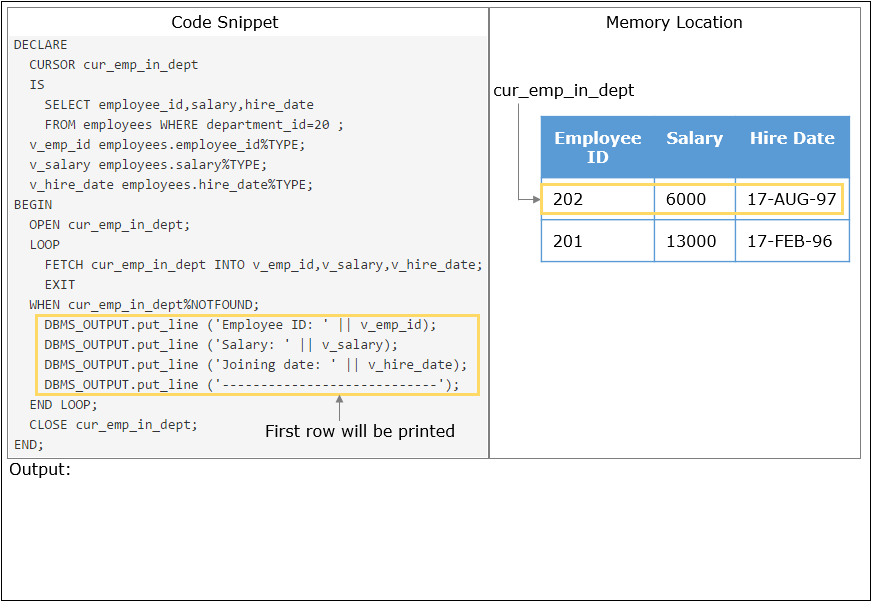


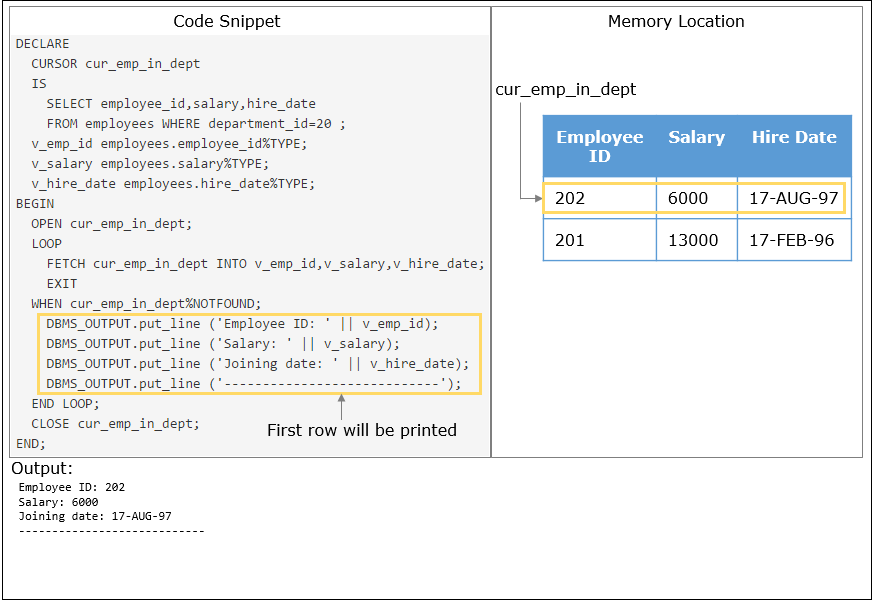


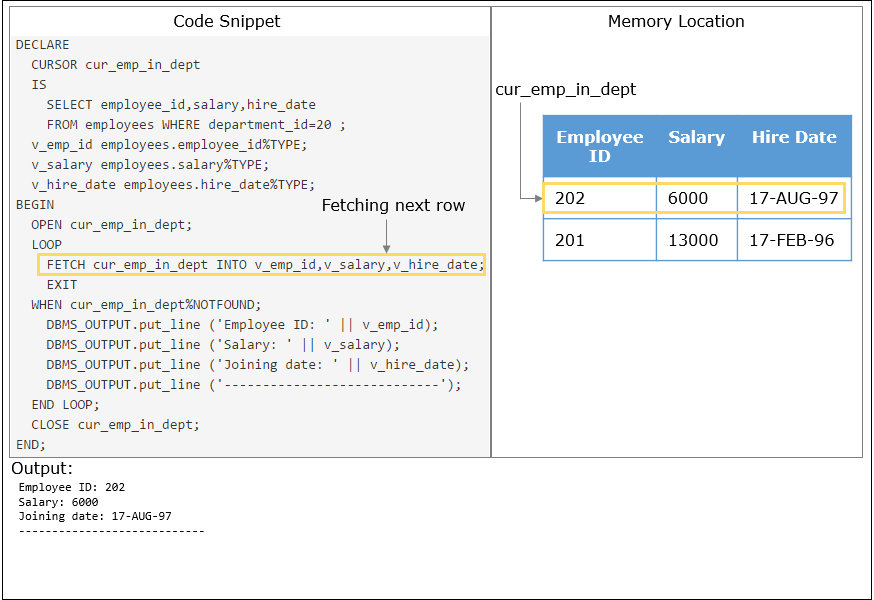


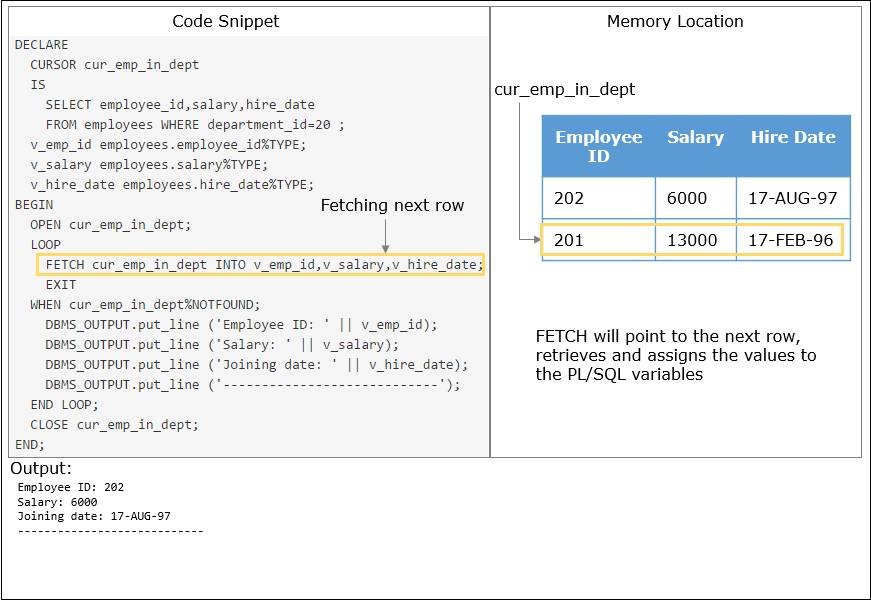


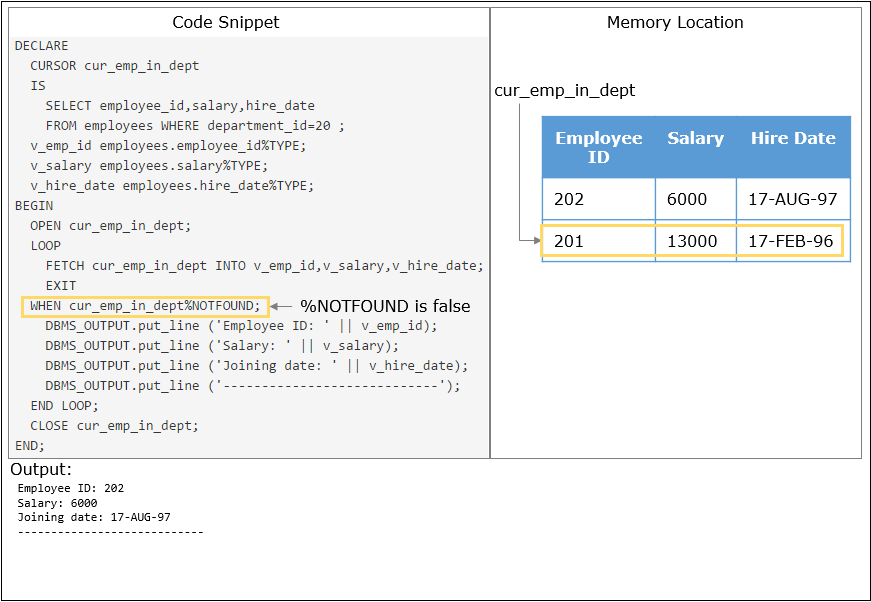


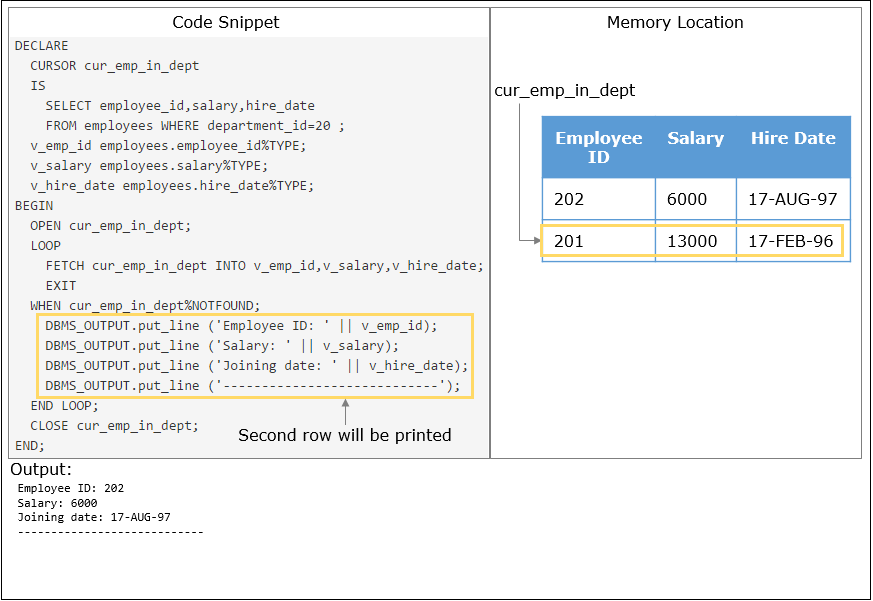


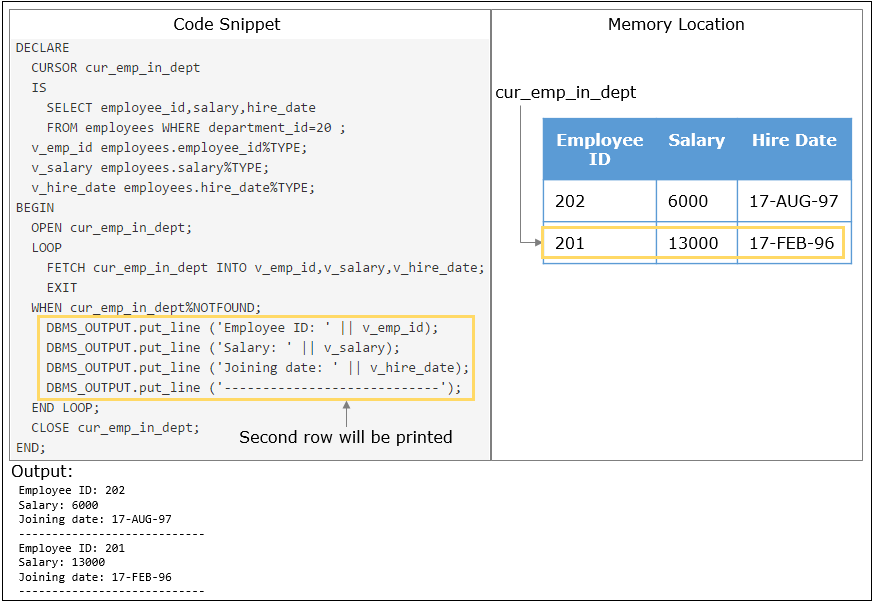


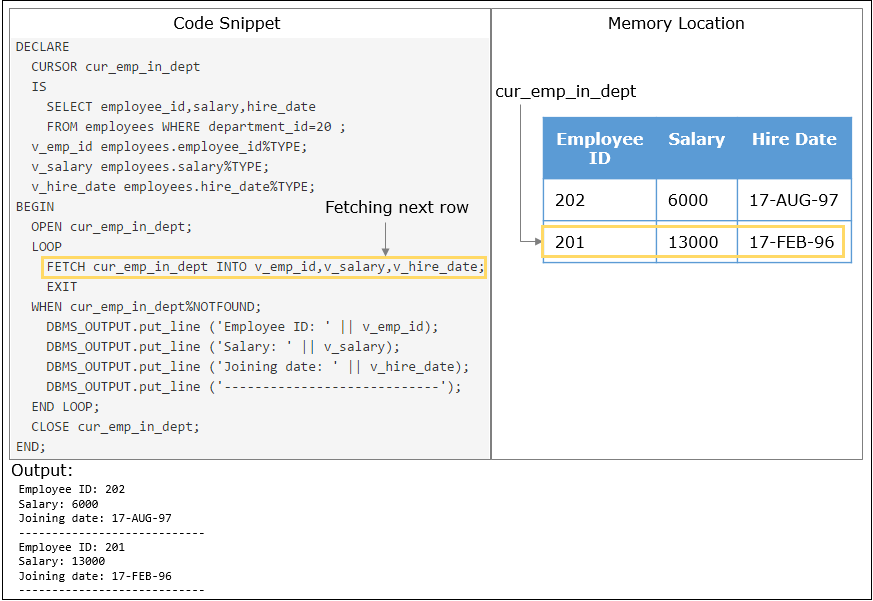


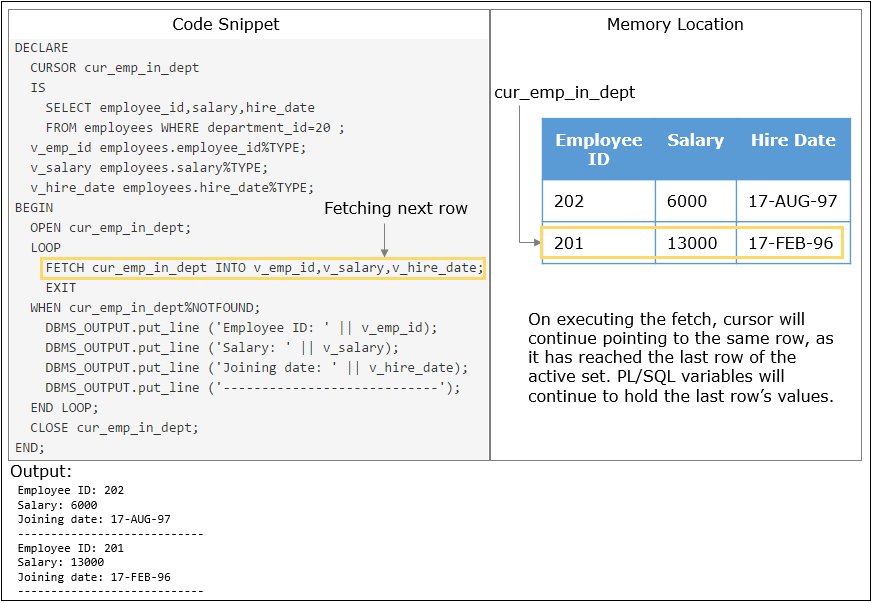


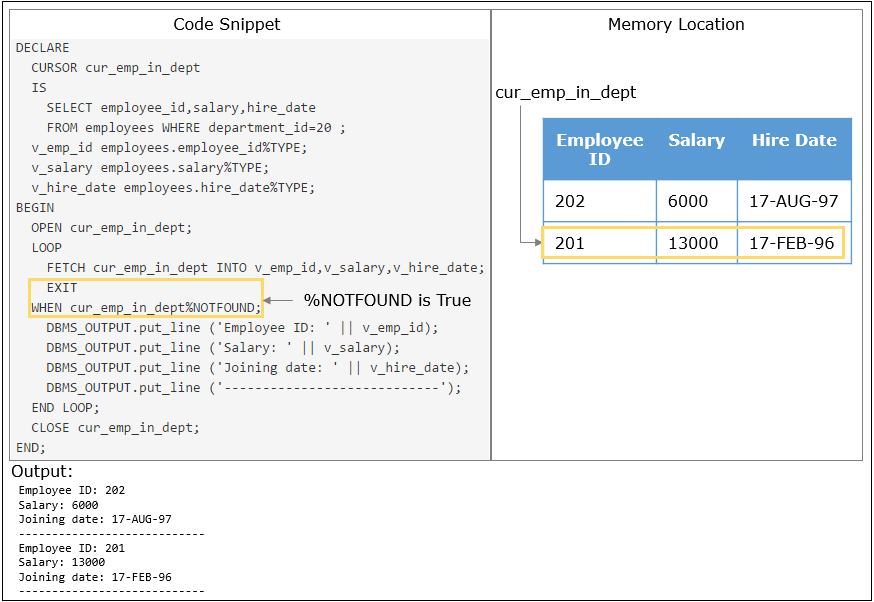


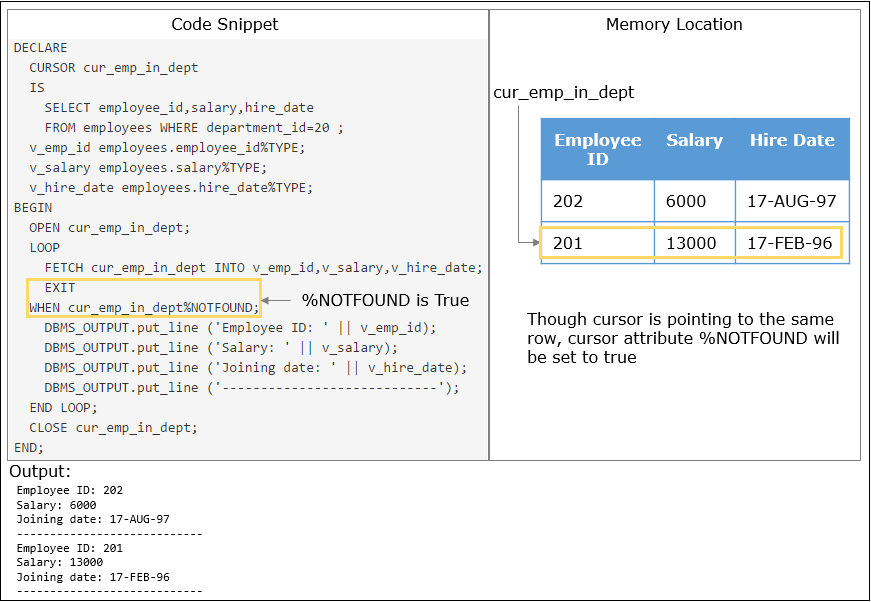


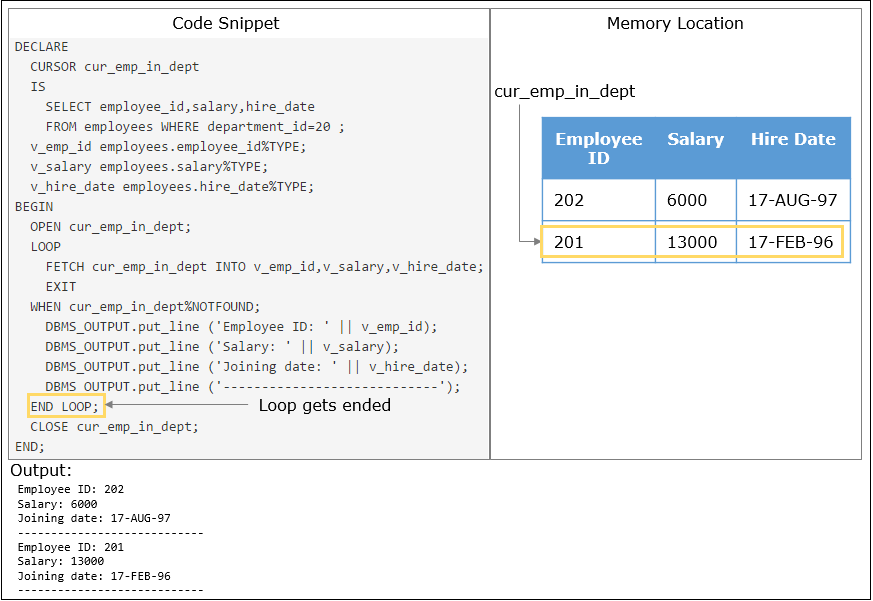


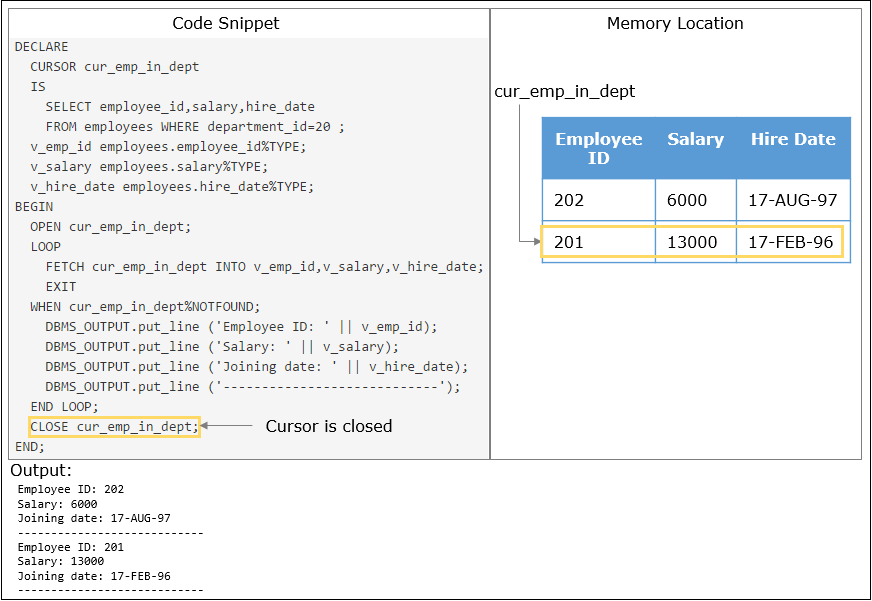


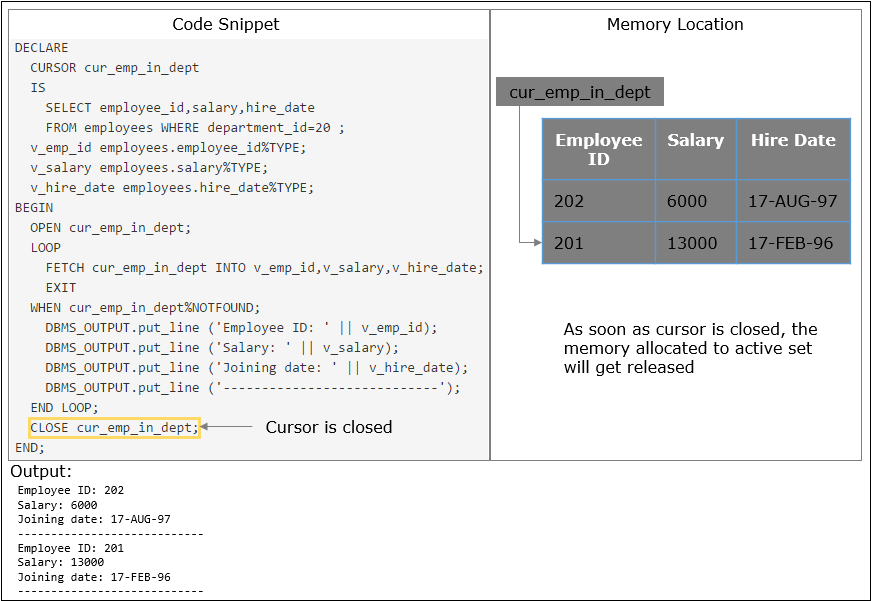












Explicit cursor is used to process multiple rows in a PL/SQL program.

A query will return a set of rows which is stored in a context area. Explicit cursors are used for accessing and processing the records from this active set, one by one. The cursor points to the current row in the active set.

Declaring an explicit cursor involves giving it a name and associating it with a query. Then you can access the query result set in either of these ways:

* Open the explicit cursor (with the OPEN statement), fetch rows from the result set (with the FETCH statement), and close the explicit cursor (with the CLOSE statement)
* Use the explicit cursor in a cursor FOR LOOP statement

# Declaring the Cursor:

Declare a cursor with CURSOR statement and associate it with a specific query.

1. CURSOR cursor\_name IS <SELECT statement>

**Note:**

* It can be any valid SELECT statement
* Do not include INTO clause in  cursor declaration because it appears later in FETCH statement
* If processing of rows in a specific sequence is required, use ORDER BY clause in the query

# Opening the Cursor:

OPEN statement executes the query associated with cursor, identifies result set, and positions cursor pointer before first row of the query result.

1. OPEN cursor\_name;

If the query returns no rows or more than one row, when the cursor is opened, PL/SQL does not raise an exception. However, you can test the status of the cursor after a fetch using the cursor attributes (will be discussed in detail later).

**Fetching cursor data:**

After cursor is opened, you can retrieve data from cursor by using FETCH statement.

1. FETCH <cursor\_name> INTO PL/SQL variables;

We need to ensure that the number of variables in the INTO clause of the FETCH statement are exactly the same as the number of columns in the SELECT query.

Care needs to be taken that the datatypes of the variables are compatible with the datatypes of the corresponding columns being fetched.

When a row is fetched, the following occurs:

* The cursor pointer advances to the next row in the identified set
* The cursor reads the data for that row and stores the values into the PL/SQL variables

# Closing the Cursor:

The CLOSE statement disables the cursor, and the active set becomes undefined, freeing its resources for reuse. Close the cursor after completing the processing of the SELECT statement. Do not attempt to fetch data from a cursor after it has been closed, or the INVALID\_CURSOR exception will be raised.

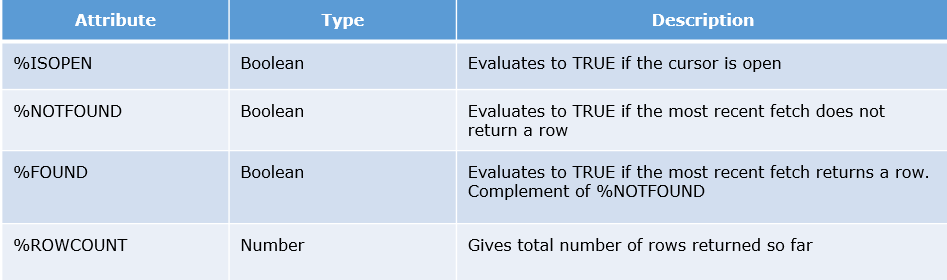
After closing an explicit cursor, you can reopen it with the OPEN statement. You must close an explicit cursor before reopening it.

1. CLOSE cursor\_name;

**Note:**

Although it is possible to terminate the PL/SQL block without closing cursors, you should make it a habit to close any cursor that you declare explicitly to free up resources.

These four attributes will help you obtain the status information of the cursor execution.



Basic information about these attributes are given, you can refer here for more on [Cursor attributes](https://docs.oracle.com/cd/B28359_01/appdev.111/b28370/cursor_attribute.htm).

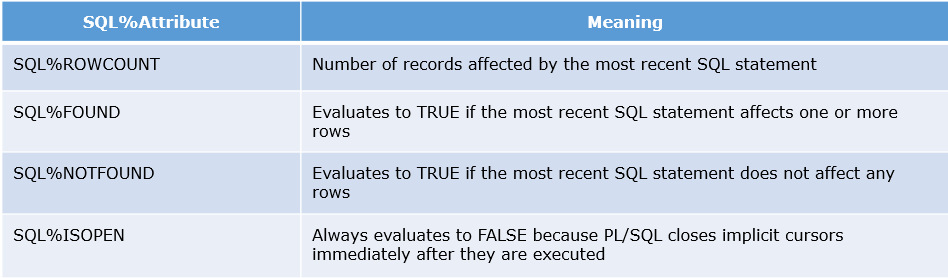
# Requirement 47:  HR Manager wants to view details of all the employees working in specific department

# Cursor records:

You have already seen that you can define records, that have the structure of columns in a table using %ROWTYPE. You can also define a record based on the selected list of columns in an explicit cursor. This is convenient for processing the rows of the active set, because you can simply fetch into the record. Therefore, the values of the row are loaded directly into the corresponding fields of the record.

**Revisiting Requirement 47,** where HR manager wants to see the details (employee ID, salary and hire date) of all the employees working in a specific department. You can use a record instead of separate variables.

# Execute the below existing program in SQL Developer and observe:

1. DECLARE
2. v\_department\_id employees.department\_id%TYPE;
3. CURSOR cur\_emp\_in\_dept
4. IS
5. SELECT employee\_id,salary,hire\_date FROM employees WHERE department\_id = v\_department\_id;
6. **--record variable of cursor rowtype is declared**
7. **--this record will have fields same as the columns in the select clause of the cursor query**
8. **--cursor record variables have to be declared after the cursor declaration**
9. **rec\_empInfo cur\_emp\_in\_dept%ROWTYPE;**
10. BEGIN
11. v\_department\_id := 20;
12. OPEN cur\_emp\_in\_dept;
13. LOOP
14. **-- values are fetched into the record, i.e. into the corresponding fields of the record**
15. FETCH cur\_emp\_in\_dept INTO **rec\_empInfo**;
16. EXIT
17. WHEN cur\_emp\_in\_dept%NOTFOUND;
18. **-- values retrieved from the corresponding fields of the record**
19. **DBMS\_OUTPUT.PUT\_LINE ('Employee ID: ' || rec\_empInfo.employee\_id||' Salary: ' || rec\_empInfo.salary||**
20. **' Joining date: ' || rec\_empInfo.hire\_date);**
21. END LOOP;
22. CLOSE cur\_emp\_in\_dept;
23. END;
24. A SQL (implicit) cursor is opened by the database to process each SQL statement that is not associated with an explicit cursor. Every SQL (implicit) cursor has attributes, each of which returns useful information about the execution of a data manipulation statement.
26. 
27. The values of cursor attributes always refer to the most recently executed SQL statement, wherever that statement appears. If you want to save an attribute value for later use, assign it to a variable immediately.
28. You can refer here for more on [SQL attributes](https://docs.oracle.com/cd/B28359_01/appdev.111/b28370/sql_cursor.htm).

Packages:

# Requirement 60: Infospark wants to create an Employee Directory System (EDS)

To organize the programs and make it available for reuse, InfoSpark has decided to create an Employee Directory System (EDS) in which employees will have the below features.

* An employee can search and view the manager's email of any other employee
* An employee can update his/her phone number

But wait, these functionalities are already developed for the EMS system! Can we use them for other applications? Yes, but these are individual programs; identifying and reusing them is a little tedious task!

So, **you can use packages to group logically related programs together**. This is of great help in application development as it makes code easy to locate, understand and reuse.

# Requirement 60: Infospark wants to create an Employee Directory System (EDS)

# Execute the below PL/SQL code to create the Package Specification:

1. CREATE OR REPLACE PACKAGE pkg\_Employee\_Info
2. IS
3. --function header defining the name, paramenter list and return type
4. FUNCTION get\_Manager\_Email(p\_empID EMPLOYEES.EMPLOYEE\_ID%TYPE) RETURN VARCHAR2;
6. --procedure header defining the name and paramenter list
7. PROCEDURE update\_Employee\_PhoneNumber(
8. p\_empID EMPLOYEES.EMPLOYEE\_ID%TYPE,
9. p\_New\_Phone\_Number EMPLOYEES.PHONE\_NUMBER%TYPE,
10. p\_status OUT NUMBER);
12. END

# Execute the below PL/SQL code to create the Package Body:

1. CREATE OR REPLACE PACKAGE BODY pkg\_Employee\_Info
2. IS
3. -- Definition of the function declared in the specification
4. FUNCTION get\_Manager\_Email(p\_empID EMPLOYEES.EMPLOYEE\_ID%TYPE)
5. RETURN VARCHAR2
6. IS
7. v\_ManagerEmail EMPLOYEES.EMAIL%TYPE;
8. BEGIN
9. SELECT email INTO v\_ManagerEmail
10. FROM employees WHERE employee\_id =(SELECT manager\_id FROM employees WHERE employee\_id = p\_empID);
11. RETURN v\_ManagerEmail;
12. EXCEPTION
13. WHEN OTHERS THEN
14. RETURN NULL;
15. END;
17. -- Definition of the procedure declared in the specification
18. PROCEDURE update\_Employee\_PhoneNumber(
19. p\_empID EMPLOYEES.EMPLOYEE\_ID%TYPE,
20. p\_New\_Phone\_Number EMPLOYEES.PHONE\_NUMBER%TYPE,
21. p\_status OUT NUMBER)
22. IS
23. v\_ManagerEmail EMPLOYEES.EMAIL%TYPE;
24. BEGIN
25. UPDATE employees
26. SET PHONE\_NUMBER = p\_New\_Phone\_Number
27. WHERE employee\_id = p\_empID;
28. p\_status := 0;
29. EXCEPTION
30. WHEN OTHERS THEN
31. p\_status := -1;
32. END;
34. END;

You can reference the package contents from other anonymous blocks, stored subprograms, database triggers and various Oracle tools.

Package name followed by . (DOT) operator, "package\_name.element", is used to access the package contents.

# Execute and observe the below PL/SQL block to reference the package elements:

1. DECLARE
2. v\_empID EMPLOYEES.EMPLOYEE\_ID%TYPE := 103;
3. v\_ManagerEmail EMPLOYEES.EMAIL%TYPE;
4. v\_status NUMBER(1);
6. BEGIN
8. -- Function in the package is called by using the package name and . operator
9. v\_ManagerEmail := **pkg\_Employee\_Info.get\_Manager\_Email(v\_empid);**
10. DBMS\_OUTPUT.put\_line(v\_ManagerEmail);
12. -- Procedure in the package is called by using the package name and . operator
13. **pkg\_Employee\_Info.update\_Employee\_PhoneNumber(v\_empID,'590.423.4570',v\_status);**
15. if v\_status = 0 then
16. DBMS\_OUTPUT.put\_line('Phone number of '||v\_empID||' is updated successfully');
17. else
18. DBMS\_OUTPUT.put\_line('Phone number of '||v\_empID||' is NOT updated successfully');
19. end if;
21. END;

ill now, you have created subprograms (procedure and function) that are stored in the database.These are called standalone subprograms. If you want to group subprograms which are logically related,you have to go for packages.

**A package is a schema object that groups logically related PL/SQL types, variables, and subprograms.**

One thing to remember is that a package itself is not any kind of executable piece of code. Instead, it is a repository for code that is executed or used. When you use a package, you actually execute or make reference to an element in a package.

**Package Specification:**

To use a package you must know what is defined and available inside the package, this information is contained in the package specification.

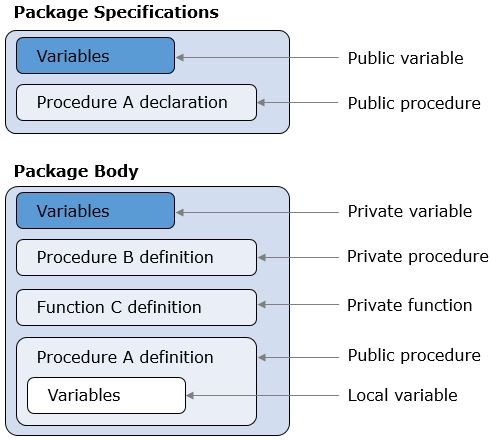
It contains the definition or specification of all the elements, (types, variables, constants, cursors, and subprograms), in the package that may be referenced outside of the package. These are called the public elements of the package.

The package specification contains all the code needed for a developer to understand how to call the objects in the package.

**Package Body:**

The package body should contain the implementation of every subprogram declared in the package specification. Subprograms defined in a package body are accessible outside the package only if their specs also appear in the package specification.

The package body can also contain types, variables and subprograms necessary for the internal workings of the package. These are called private elements which can be referenced only within the package body and not outside the package.



# Syntax for package specification:

1. CREATE [OR REPLACE] PACKAGE pkg\_package\_name
2. IS|AS
3. [constant\_declaration ...]
4. [variable\_declaration ...]
5. [cursor\_spec ...]
6. [function\_spec ...]
7. [procedure\_spec ...]
8. END [pkg\_package\_name];

# Syntax for package body:

1. CREATE [OR REPLACE] PACKAGE BODY pkg\_package\_name
2. IS|AS
3. [constant\_usage ...]
4. [exception\_usage ...]
5. [variable\_usage ...]
6. [cursor\_body ...]
7. [function\_body ...]
8. [procedure\_body ...]
9. END [pkg\_package\_name];

A package specification can exist without a package body, called [bodiless package](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/packages.htm#i2412), but the package body cannot exist without a package specification.