SQL/PL-SQL Booklet

DOMDOUZIS, Konstantinos <http://orcid.org/0000-0003-3679-3527>

SQL/PL-SQL Booklet

Dr Konstantinos Domdouzis

Department of Computing
Sheffield Hallam University

Structured-Query Language (SQL)

- Management of data included in the Tables of a RDBMS
- **Why is it called Structured?**
- It is an ANSI and ISO standard for RDBMSs, and virtually all RDBMSs use it.
- SQL is largely ‘declarative’ (you say what you want, not how to do it)

Structured-Query Language (SQL)

- **Data Definition Language (DDL):**
  statements are used to define the database structure or schema

- **Data Manipulation Language (DML):**
  statements are used for managing data within schema objects.

- **Data Control Language (DCL):**
  statements that provide or withdraw data access rights.

- **Transaction Control (TCL):**
  statements are used to manage the changes made by DML statements.

Structured-Query Language (SQL)

CLAUSE UPDATE Customers

CLAUSE SET ContactName='Alfred'

EXPRESSION

CLAUSE WHERE CustomerName='John';

PREDICATE

SQL STATEMENT

SQL Syntax

- **SQL Statements**

CREATE TABLE
SELECT
DROP TABLE
CREATE INDEX
DROP INDEX
DESC
TRUNCATE TABLE
ALTER TABLE
ALTER TABLE (RENAME)
INSERT INTO
UPDATE
DELETE
CREATE DATABASE
DROP DATABASE
USE
COMMIT
ROLLBACK

SQL Syntax

- **SQL Reserved Words**

WHERE
DISTINCT
AND/OR
IN
BETWEEN
LIKE
ORDER BY
GROUP BY
COUNT
HAVING

Exact Numeric Data Types

DATA TYPE	FROM	TO
bigint	-9,223,372,036,854,775,808	9,223,372,036,854,775,807
int	-2,147,483,648	2,147,483,647
smallint	-32,768	32,767
tinyint	0	255
bit	0	1
decimal	-10^38 +1	10^38 -1
numeric	-10^38 +1	10^38 -1
money	-922,337,203,685,477.5808	+922,337,203,685,477.5807
smallmoney	-214,748.3648	+214,748.3647

Approximate Numeric Data Types

DATA TYPE	FROM	TO
float	-1.79E + 308	1.79E + 308
real	-3.40E + 38	3.40E + 38

Date and Time Data Types

DATA TYPE	FROM	TO
datetime	Jan 1, 1753	Dec 31, 9999
smalldatetime	Jan 1, 1900	Jun 6, 2079
date	Stores a date like June 30, 1991	
time	Stores a time of day like 12:30 P.M.	

Character Strings Data Types

DATA TYPE	FROM	TO
char	char	Maximum length of 8,000 characters.(Fixed length non-Unicode characters)
varchar	varchar	Maximum of 8,000 characters.(Variable-length non-Unicode data).
varchar(max)	varchar(max)	Maximum length of 231characters, Variable-length non-Unicode data (SQL Server 2005 only).
text	text	Variable-length non-Unicode data with a maximum length of 2,147,483,647 characters.

Unicode Character Strings Data Type

DATA TYPE	Description
nchar	Maximum length of 4,000 characters.(Fixed length Unicode)
nvarchar	Maximum length of 4,000 characters.(Variable length Unicode)
nvarchar(max)	Maximum length of 231characters (SQL Server 2005 only).(Variable length Unicode)
ntext	Maximum length of 1,073,741,823 characters. (Variable length Unicode)

Binary Data Type

DATA TYPE	Description
binary	Maximum length of 8,000 bytes(Fixed-length binary data)
varbinary	Maximum length of 8,000 bytes.(Variable length binary data)
varbinary(max)	Maximum length of 231 bytes (SQL Server 2005 only). (Variable length Binary data)
image	Maximum length of 2,147,483,647 bytes. (Variable length Binary Data)

Miscellaneous Data Types

DATA TYPE	Description
sql_variant	Stores values of various SQL Server-supported data types, except text, ntext, and timestamp.
timestamp	Stores a database-wide unique number that gets updated every time a row gets updated
uniqueidentifier	Stores a globally unique identifier (GUID)
xml	Stores XML data. You can store xml instances in a column or a variable (SQL Server 2005 only).
cursor	Reference to a cursor object
table	Stores a result set for later processing

Naming Rules

Apply to tables, views and columns. Table and View names must be unique within a database. Column names must be unique within a table.

- Must start with a letter.
- May be between 1 and 30 characters long.
- May contain alphabetic and numeric characters A to Z, a to z, 0 to 9.
- They are NOT case sensitive.
- Cannot be a reserved word.
- May contain underscores**.**

CREATE Table

The basic form of the CREATE statement is:

```
CREATE  TABLE  tablename
   (column_name          datatype ,
    column_name          datatype ,
   . . . . .
    column_name          datatype
   ) ;
```

Example

```
CREATE TABLE ACC
        (AccNo          Number(7) ,
         Balance        Number(7,2) ,
         Branch         Varchar2(15) ,
         Opened         Date,
         Bonus          Number(7,2)
        ) ;
```

INSERT

The basic form of the INSERT statement is:

INSERT INTO tablename **VALUES** (...., ..., ..., ...);

INSERT INTO ACC **VALUES**
(1494315, 0.50, 'Tinsley', '01-SEP-2003'**,** 1.00);

- note that values for all columns are specified.

But also,

INSERT INTO ACC (AccNo, Balance, Branch, Opened)
VALUES (1245890, 234.50, 'BROOMHILL‘,'12-Nov-03');

- here values are given only for the specified columns.

One insert statement is required for **each** row to be inserted!

UPDATE

The basic form of the UPDATE statement is:

UPDATE tablename
SET column_name = newvalue;

- but this would update every row !

A better form is

UPDATE tablename
SET column_name = newvalue
WHERE column_name = testvalue;

- this will only update rows which satisfy the WHERE clause

DELETE

- Deleting Data from Tables

The basic form of the DELETE statement is:

DELETE FROM tablename;

- but this would delete every row !

A better form is:

DELETE FROM tablename
WHERE column_name = testvalue;

- this will only update rows which satisfy the WHERE clause

DELETE

- Deleting entire Table

The basic form of the **DROP** statement is:

DROP TABLE tablename ;

DROP TABLE ACC ;

This destroys the structure and the data !

SELECT COUNT(columnname) FROM tablename;

- The COUNT(columnname) function returns the number of values (NULL values **will not** be counted) of the specified column

SELECT COUNT(*) FROM tablename;

- The function returns the number of records in a table
- What happens to the NULL values?
- The function **counts** the NULL values.

SELECT COUNT(DISTINCT columnname) FROM tablename;

- The function returns unique, non-NULL values from the column <column_name>

SELECT COUNT(DISTINCT columnname1), COUNT(DISTINCT columnname2) FROM tablename;

- The function returns unique, non-NULL values from both columns

SELECT count(*) FROM multiple tables

Example:

```
SELECT(
      SELECT COUNT(*)
      FROM   employees
      ) AS Total_Employees,
      (SELECT COUNT(*)
      FROM   departments
      ) AS No_Of_Departments
FROM dual
```

SELECT MAX(columnname) FROM tablename;

- The MAX() function returns the largest value of the selected column.
- The function takes only one argument.

SELECT MIN(columnname) FROM tablename;

- The MIN() function returns the smallest value of the selected column.
- The function takes only one argument.

Behaviour of NULL values with MAX() and MIN()

- When a column contains only NULL values, MAX() and MIN() will output NULL.

- If a column includes NULL and non-NULL numerical values, NULLs are ignored.

- If a column includes NULL and characters, NULLs are ignored, while characters are compared based on their hexadecimal value.

SELECT AVG(columnname) FROM tablename

- The AVG() returns the average value of attributes included in the selected column
- The function does not accept more than one arguments
- The function does not count NULL values

SELECT SUM(columnname) FROM tablename;

- The SUM() function returns the total sum of a numeric column.
- The function does not accept more than one arguments
- The function does not count NULL values

GROUP BY Clause

- The GROUP BY statement is used in conjunction with the aggregate functions to group the result-set by one or more columns.

```
SELECT column_name,     aggregate_function(column_name)

FROM table_name

GROUP BY column_name;
```

- The function works also with multiple columns as long as they are the same as those specified in the SELECT Clause.

```
eg. select ename, job, sum(sal) from emp
      group by ename, job;
```

HAVING Clause

- The HAVING clause was added to SQL because the WHERE clause could not be used with aggregate functions.

```
SELECT column_name,   aggregate_function(column_name)

FROM table_name

WHERE column_name operator value

GROUP BY column_name

HAVING aggregate_function(column_name)  operator value;
```

Primary Keys

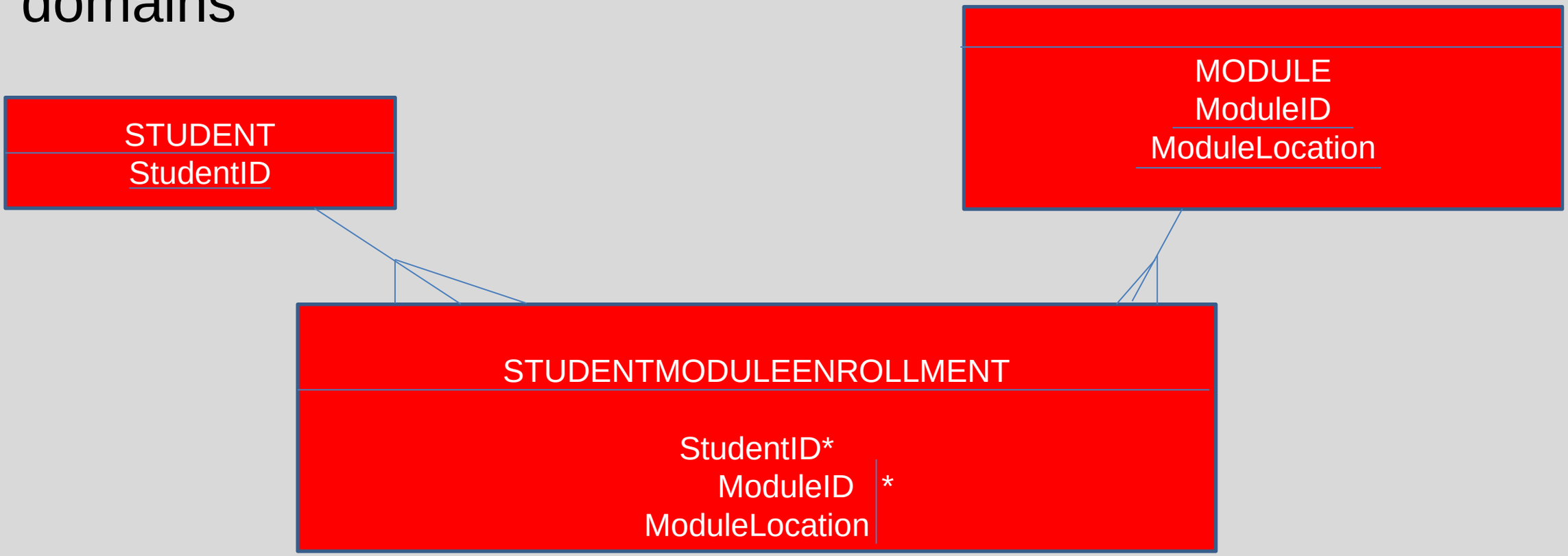
- A primary key is a field(column) in a table which **uniquely** identifies each record (row) in a database table

- A primary key column **cannot** have NULL values

- A database table can only have **ONE and ONLY ONE** Primary Key

Compound Keys

- A **COMPOUND** Key is a key that includes two or more domains



How to create Primary Keys

- Let's say I want to create a table that will include data about football players. How do I do it?

```
CREATE TABLE FOOTBALLPLAYERS(

        ID   INT  NOT NULL,
        NAME VARCHAR (20) NOT NULL,
        AGE  INT  NOT NULL,
        ADDRESS  CHAR (25) ,

        PRIMARY KEY (ID)
);
```

Or we can create the Primary Key like this

```
CREATE TABLE FOOTBALLPLAYERS(

    ID   INT  NOT NULL PRIMARY KEY,
    NAME VARCHAR (20) NOT NULL,
    AGE  INT  NOT NULL,
    ADDRESS  CHAR (25) ,

);
```

How to create Compound Primary Keys

- Let's create the football players table with a compound primary key this time

```
CREATE TABLE FOOTBALLPLAYERS(

        ID   INT NOT NULL,
        NAME VARCHAR (20)     NOT NULL,
        AGE  INT NOT NULL,
        ADDRESS  CHAR (25) ,

        PRIMARY KEY (ID, NAME)

);
```

Foreign Key

- A key to link two tables together

- A Foreign Key is a column or a combination of columns whose values match a Primary Key in a different table.

DEFINITION OF CONSTRAINTS

RDBMS Data Integrity

- **Entity Integrity:** No duplicate rows in a table!!
- **Domain Integrity:** Valid entries are entered in a given column by restricting the type, the format, or the range of values.
- **Referential integrity:** Rows cannot be deleted especially when they are used by other records.
- **User-Defined Integrity:** Enforces some specific business rules that do not fall into entity, domain or referential integrity.

What is a Constraint ?

In general terms:

– a constraint is a control that limits or restricts actions or behaviour.

In database terms a CONSTRAINT is:

– a rule which limits the values that can be stored in a table column

– implemented by code stored in the database

– specified as part of the CREATE TABLE statement

– added / removed later with the ALTER TABLE statement

– better if given a meaningful name

– checked and applied whenever an INSERT, UPDATE or DELETE statement is run against that table.

Database Constraints are used to implement three different types of check designed to preserve the integrity of the data.

Entity integrity – **Referential integrity** – **Domain integrity**

Constraints

- Constraints are rules
- The limit the type of data that go into a table
- Ensure data accuracy and reliability
- Two types of constraints: Column-level, Table-level

Entity Integrity

Every entity must have a unique identifier. Each table corresponds to an entity, so each table must also have a unique identifier.

It can be a single column, - which must have a unique value and which cannot be NULL.

Or it can be a combination of columns, - which must have a unique combination of non-NULL values

CUST

RefNo	Name		
A123	**J Doe**		
A124	**J Smith**		
B127	**R Best**	**4 East Row**	**Rotherham**
B128	**J Best**		
C371	**R Done**		

CUSTACC

RefNo	AccNo
A123	**1245890**
A123	**1494315**
B127	**5418490**
B128	**5418490**

ACC

AccNo	Balance	Branch	Opened	Bonus	
1245890	**234.50**	**Broomhill 12**	**Nov 2003**	**100.00**	
1494315	**0.50**	**Tinsley 15**	**Dec 1999**	**0.00**	
5418490	**1789.40**	**Broomhill 06**	**May 1988**		

Entity Integrity

We know that these columns are the unique identifiers, but the DBMS doesn't know that, . . yet.

We need to declare the unique identifier columns(s) as the PRIMARY KEY of the table.

Then the DBMS will create an integrity check on the column(s) as a PRIMARY KEY CONSTRAINT.

For a PRIMARY KEY the check will enforce a rule that the column values must be UNIQUE and NOT NULL.

CUST

PK RefNo	Name		
A123	J Doe		
A124	J Smith		
B127	R Best		
B128	J Best		
C371	R Done	23 Middle Avenue	Barnsley

CUSTACC

PK RefNo	AccNo
A123	1245890
A123	1494315
B127	5418490
B128	5418490

ACC

PK AccNo	Balance	Branch	Opened	Bonus
1245890	234.50	Broomhill 12	Nov 2003	100.00
1494315	0.50	Tinsley 15	Dec 1999	0.00
5418490	1789.40	Broomhill 06	May 1988	

CUSTACC has only one PK (the combination of RefNo+AccNo) Not two !

Referential Integrity

The relational model relies on being able to relate tables by using columns* common to each table.

In the bank account tables:

RefNo is common to CUST and CUSTACC
AccNo is common to ACC and CUSTACCr

which allows us to relate Customers to their Accounts,
. . . provided that none of the values are missing or wrong

* It's the data in the columns which matters, not the column names.

CUST

PK RefNo	Name
A123	J Doe
A124	J Smith
B127	R Best
B128	J Best
C371	R Done

CUSTACC

PK RefNo	AccNo
A123	1245890
A123	1494315
B127	5418490
B128	5418490

ACC

PK AccNo	Balance	Branch	Opened	Bonus
1245890	234.50	Broomhill 12	Nov 2003	100.00
1494315	0.50	Tinsley 15	Dec 1999	0.00
5418490	1789.40	Broomhill 06	May 1988	

Referential Integrity

We know that the RefNo in CUSTACC refers to the PRIMARY KEY of another table (CUST), but the DBMS doesn't, . . yet.

We need to declare the referring column(s) as a FOREIGN KEY and say which table and column it REFERENCES.

CUST (RefNo) in this case

ACC (AccNo) in this case

Then the DBMS will create an integrity check on the column(s) as a FOREIGN KEY CONSTRAINT.

For a FOREIGN KEY the rule is that if the column value is not NULL, then a matching row MUST EXIST in the other table

CUST

PK RefNo	Name
A123	J Doe
A124	J Smith
B127	R Best
B128	J Best
C371	R Done

CUSTACC

PK RefNo	FK AccNo
A123	1245890
A123	1494315
B127	5418490
B128	5418490

ACC

PK AccNo	Balance	Branch	Opened	Bonus
1245890	234.50	Broomhill 12	Nov 2003	100.00
1494315	0.50	Tinsley 15	Dec 1999	0.00
5418490	1789.40	Broomhill 06	May 1988	

Domain Integrity

Is concerned with ensuring that column values conform to ‘business rules’.

Various types of constraint are available for columns:

NOT NULL - prevents a column value of NULL

UNIQUE - checks that the column value is unique for this column in the table

CHECK <condition>

- checks that column values satisfy the condition

DEFAULT <value>

- sets a default value if it is not specified in an INSERT

CUST

PK RefNo	Name
A123	J Doe
A124	J Smith
B127	R Best
B128	J Best
C371	R Done

CUSTACC

PK RefNo	FK AccNo
A123	1245890
A123	1494315
B127	5418490
B128	5418490

ACC

PK AccNo	Balance	Branch	Opened	Bonus
1245890	234.50	Broomhill 12	Nov 2003	100.00
1494315	0.50	Tinsley 15	Dec 1999	0.00
5418490	1789.40	Broomhill 06	May 1988	

Domain Integrity

In the accounting system we might decide to:

default the account opening date to ‘Today’
Opened DATE DEFAULT SYSDATE

apply a check on the bonus
Bonus NUMBER(7,2) CHECK(Bonus <500.00)

make the customer name not nullable
Name VARCHAR2(25) NOT NULL

CUST

PK RefNo	Name		
A123	J Doe		
A124	J Smith		
B127	R Best		
B128	J Best		
C371	R Done	23 Middle Avenue	Barnsley

CUSTACC

PK RefNo	FK AccNo
A123	1245890
A123	1494315
B127	5418490
B128	5418490

ACC

PK AccNo	Balance	Branch	Opened	Bonus
1245890	234.50	Broomhill 12	Nov 2003	100.00
1494315	0.50	Tinsley 15	Dec 1999	0.00
5418490	1789.40	Broomhill 06	May 1988	

Domain Integrity

We don't need to apply NOT NULL and UNIQUE constraints to the unique identifier columns (RefNo, RefNo+AccNo, AccNo)

. . . the PRIMARY KEY constraint does that.

CUST

PK RefNo	Name		
A123	J Doe	1 High Street	Sheffield
A124	J Smith	2 West Street	Sheffield
B127	R Best	4 East Row	Rotherham
B128	J Best	4 East Row	Rotherham
C371	R Done	23 Middle Avenue	Barnsley

CUSTACC

PK RefNo	FK AccNo
A123	1245890
A123	1494315
B127	5418490
B128	5418490

ACC

PK AccNo	Balance	Branch	Opened	Bonus
1245890	234.50	Broomhill 12	Nov 2003	100.00
1494315	0.50	Tinsley 15	Dec 1999	0.00
5418490	1789.40	Broomhill 06	May 1988	

CREATION OF CONSTRAINTS

Examples of Constraints

PRIMARY Key: Uniquely identified each rows/records in a database table.

FOREIGN Key: Uniquely identified a rows/records in any another database table.

NOT NULL Constraint: Ensures that a column cannot have NULL value.

DEFAULT Constraint: Provides a default value for a column when none is specified.

UNIQUE Constraint: Ensures that all values in a column are different.

CHECK Constraint: The CHECK constraint ensures that all values in a column satisfy certain conditions.

INDEX: Used to create and retrieve data from the database very quickly.

The PRIMARY KEY Constraint

In order to allow naming of a Primary Key constraint and to define the primary key in multiple columns, we use the following syntax:

```
CREATE TABLE Persons
(
P_Id int NOT NULL,
LastName varchar(255) NOT NULL,
FirstName varchar(255),
Address varchar(255),
CONSTRAINT pk_PersonID PRIMARY KEY (P_Id,LastName)
)
```

The FOREIGN KEY Constraint

Imagine we have the following tables:

P_ID	FirstName	LastName
1	John	Davis
2	Nick	Benson
3	Louise	Hodgson

Persons Table

O_ID	OrderName
1	Shirt
2	Jacket
3	Dress

Orders Table

The FOREIGN KEY Constraint

```
CREATE TABLE Orders
(
O_Id int NOT NULL PRIMARY KEY,
OrderNo int NOT NULL,
FOREIGN KEY (P_Id) REFERENCES Persons(P_Id)
)
```

The FOREIGN KEY Constraint

Now the tables are connected to each other:

P_ID	FirstName	LastName
1	John	Davis
2	Nick	Benson
3	Louise	Hodgson

Persons Table

O_ID	OrderName	P_ID
1	Shirt	2
2	Jacket	1
3	Dress	3

Orders Table

The FOREIGN KEY Constraint

In order to allow naming of a FOREIGN KEY Constraint and to define a FOREIGN KEY Constraint on multiple columns, we use the following syntax:

```
CREATE TABLE Orders
(
O_Id int NOT NULL,
OrderNo int NOT NULL,
P_Id int,
PRIMARY KEY (O_Id),
CONSTRAINT fk_PerOrders FOREIGN KEY (P_Id)
REFERENCES Persons(P_Id)
)
```

The NOT NULL Constraint

Example - Notice the NOT NULL Constraint next to the AGE Column

```
CREATE TABLE FootballPlayers(
        ID   INT  NOT NULL,
        NAME VARCHAR (20) NOT NULL,
        AGE  INT NOT NULL,
        ADDRESS  CHAR (25) ,
        PRIMARY KEY (ID)
);
```

The NOT NULL Constraint

The NOT NULL Constraint can also be applied to an attribute of VARCHAR Data Type

Example

```
CREATE TABLE FootballPlayers
(Customer_ID integer NOT NULL,
Last_Name varchar (30) NOT NULL,
First_Name varchar(30));
```

The DEFAULT Constraint

The DEFAULT Constraint provides a default value to a column when an INSERT INTO statement does not provide a specific value.

Example

```
CREATE TABLE FootballPlayers(
        ID   INT NOT NULL,
        NAME VARCHAR (20) NOT NULL,
        AGE  INT NOT NULL,
        ADDRESS  CHAR (25) DEFAULT '153 Arundel Street',
        SALARY   DECIMAL (18, 2) DEFAULT 5000.00,
        PRIMARY KEY (ID)
);
```

The UNIQUE Constraint

The UNIQUE Constraint prevents two records from having identical values at a specific column.

Example

```
CREATE TABLE FootballPlayers(
        ID INT NOT NULL,
        NAME VARCHAR (20) NOT NULL,
        AGE  INT NOT NULL UNIQUE,
        ADDRESS  CHAR (25) ,
        PRIMARY KEY (ID)
);
```

The UNIQUE Constraint

To allow naming of a UNIQUE Constraint and for defining a UNIQUE Constraint to multiple columns, we use the following syntax:

```
CREATE TABLE Persons
(
P_Id int NOT NULL,
LastName varchar(255) NOT NULL,
FirstName varchar(255),
Address varchar(255),
CONSTRAINT uc_PersonID UNIQUE (P_Id,LastName)
)
```

The CHECK Constraint

The CHECK Constraint prevents two records from having identical values at a specific column.

Example

```
CREATE TABLE FootballPlayers(
        ID   INT NOT NULL,
        NAME VARCHAR (20) NOT NULL,
        AGE  INT NOT NULL CHECK (AGE >= 18),
        ADDRESS  CHAR (25) ,
        PRIMARY KEY (ID)
);
```

The CHECK Constraint

In order to allow naming of a CHECK constraint and in order to use it in multiple columns, we use the following syntax:

```
CREATE TABLE Persons
(
P_Id int NOT NULL,
LastName varchar(255) NOT NULL,
FirstName varchar(255),
Address varchar(255),
CONSTRAINT chk_Person CHECK (P_Id>0 AND City='Sheffield')
)
```

The INDEX Constraint

- The INDEX Constraint is used to retrieve data from the database very quickly.

- Indexes can be created by using a single or a group of columns in a table.

- Indexes are good for large databases.

So what exactly is an Index?

- An index is similar to a book index
- Instead of checking the full book (full scan) for a specific value, we refer immediately to the index.
- So an index is a data structure that stores the values for a specific column of a table.

So what exactly is an Index?

- An index is similar to a book index
- Instead of checking the full book (full scan) for a specific value, we refer immediately to the index.
- So an index is a data structure that stores the values for a specific column of a table.

How to create an Index?

```
CREATE INDEX index_name
ON table_name (column_name)
```

```
CREATE INDEX index_name
ON table_name (column_name1, column_name2)
```

THE SQL ALTER

SQL ALTER TABLE

It is used to add, delete or modify columns in an existing table

It is used to add or drop various constraints on an existing table

Add a column using ALTER

```
ALTER TABLE table_name ADD column_name datatype;
```

Add multiple columns using ALTER

```
ALTER TABLE table_name ADD column_name1 datatype,
column_name2 datatype,
column_name3 datatype;
```

Drop a column using ALTER

ALTER TABLE table_name DROP COLUMN column_name;

Change the data type of a column using ALTER

ALTER TABLE table_name MODIFY COLUMN column_name datatype;

ALTER & CONSTRAINTS

Add a NOT NULL Constraint to a column using ALTER

ALTER TABLE table_name MODIFY column_name datatype NOT NULL;

Add a UNIQUE Constraint using ALTER

ALTER TABLE table_name
ADD CONSTRAINT ConstraintName UNIQUE(column1, column2...);

Add a CHECK Constraint using ALTER

```
ALTER TABLE table_name
ADD CONSTRAINT ConstraintName
CHECK (CONDITION);
```

Add a PRIMARY KEY Constraint using ALTER

```
ALTER TABLE table_name
ADD CONSTRAINT PrimaryKeyName PRIMARY KEY
(column1, column2...);
```

Drop a Constraint using ALTER

```
ALTER TABLE table_name
DROP CONSTRAINT ConstraintName;
```

Drop a PRIMARY KEY Constraint using ALTER

```
ALTER TABLE table_name
DROP CONSTRAINT PrimaryKeyName;
```

Foreign Key Constraints & ALTER

Imagine we have the following tables:

P_ID	FirstName	LastName
1	John	Davis
2	Nick	Benson
3	Louise	Hodgson

Persons Table

O_ID	OrderName
1	Shirt
2	Jacket
3	Dress

Orders Table

Foreign Key Constraints & ALTER

When a table has been created and we want to create a Foreign Key constraint on one of its columns, we use the following SQL statement:

ALTER TABLE Orders
ADD FOREIGN KEY (P_Id)
REFERENCES Persons(P_Id)

However, before we use the above SQL statement, we must have created the column P_ID already in the Orders Table. The above statement references the column to the Primary Key of Persons. Furthermore, the above statement creates the constraint to the Orders Table. We need to insert the values for the P_Id column in the Orders Table using the INSERT INTO Statement.

Foreign Key Constraints & ALTER

Now the tables are connected to each other:

Persons Table

P_ID	FirstName	LastName
1	John	Davis
2	Nick	Benson
3	Louise	Hodgson

Orders Table

O_ID	OrderName	P_ID
1	Shirt	2
2	Jacket	1
3	Dress	3

Foreign Key Constraints & ALTER

In order to name a Foreign Key constraint or in order to define a Foreign Key constraint to multiple columns, we use the following SQL statements:

```
ALTER TABLE Orders
ADD CONSTRAINT fk_PerOrders
FOREIGN KEY (P_Id)
REFERENCES Persons(P_Id)
```

Drop Foreign Key Constraints using ALTER

```
ALTER TABLE Orders
DROP CONSTRAINT fk_PerOrders
```

The above SQL statement used for a constraint which has a specific name that we gave. What if our constraint is nameless? In the case of Oracle PL/SQL, the software assigns automatically a name to the Primary & Foreign Key constraints.

JOINS

Definition of JOINS

The SQL Joins clause is used to combine records from two or more tables in a database.

A JOIN is a means for combining fields from two tables by using values common to each.

Types of SQL JOIN

INNER JOIN

LEFT JOIN

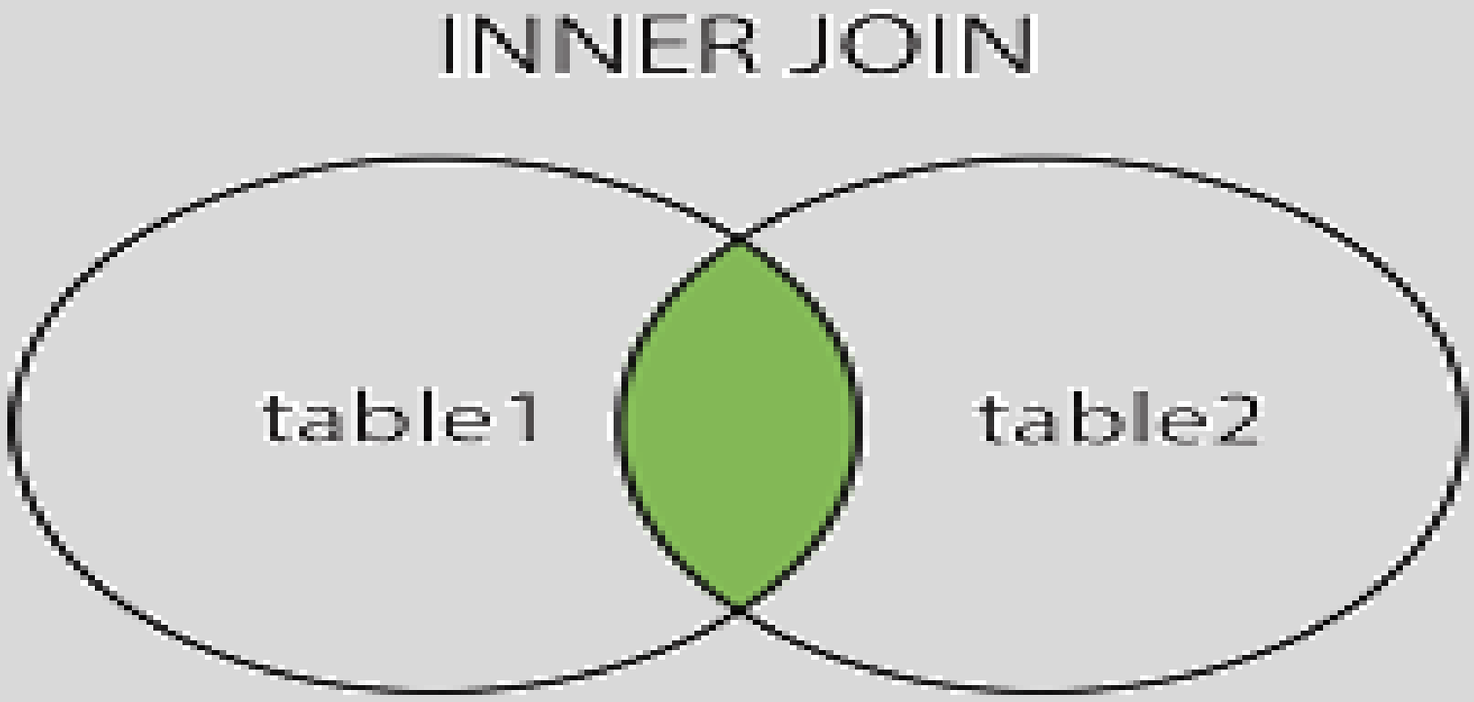
RIGHT JOIN

FULL JOIN

SELF JOIN

CARTESIAN JOIN

SQL INNER JOIN



SQL INNER JOIN Syntax

SELECT *column_name(s)*

FROM *table1*

INNER JOIN *table2*

ON *table1.column_name=table2.column_name*;

Example of SQL INNER JOIN

Imagine we have the following tables:

P_ID	FirstName	LastName
1	John	Davis
2	Nick	Benson
3	Louise	Hodgson

Persons Table

O_ID	OrderName	P_ID
1	Shirt	2
2	Jacket	1
3	Dress	3

Orders Table

Example of SQL INNER JOIN

```
SELECT Persons.FirstName, Orders.OrderName

FROM Persons

INNER JOIN Orders

ON Persons.P_ID=Orders.P_ID
```

Example of SQL INNER JOIN

The result of the INNER JOIN is the following:

FirstName	OrderNo
Benson	Shirt
Davis	Jacket
Hodgson	Dress

Example of SQL INNER JOIN

What if I had the following two tables:

P_ID	FirstName	LastName
1	John	Davis
2	Nick	Benson
3	Louise	Hodgson

Persons Table

O_ID	OrderName	P_ID
1	Shirt	2
2	Jacket	1
3	Dress	2

Orders Table

(I changed the value of P_ID to 2 instead of 3 in row with O_ID=3)

Example of SQL INNER JOIN

```
SELECT Persons.FirstName, Orders.OrderName

FROM Persons

INNER JOIN Orders

ON Persons.P_ID=Orders.P_ID
```

Example of SQL INNER JOIN

Now the result of the INNER JOIN is the following:

FirstName	OrderNo
Benson	Shirt
Davis	Jacket
Benson	Dress

SQL LEFT JOIN

The LEFT JOIN keyword returns all rows from the left table (table1), with the matching rows in the right table (table2). The result is NULL in the right side when there is no match.

SQL LEFT JOIN

SELECT *column_name(s)*

FROM *table1*

LEFT JOIN *table2*

ON *table1.column_name=table2.column_name*;

Example of SQL LEFT JOIN

Imagine we have the following tables:

P_ID	FirstName	LastName
1	John	Davis
2	Nick	Benson
3	Louise	Hodgson

Persons Table

O_ID	OrderName	P_ID
1	Shirt	2
2	Jacket	2
3	Dress	3

Orders Table

Example of SQL LEFT JOIN

```
SELECT Persons.P_ID, Persons.FirstName,
Persons.LastName, Orders.OrderName

FROM Persons

LEFT JOIN Orders

ON Persons.P_ID=Orders.P_ID
```

Example of SQL LEFT JOIN

The result of the LEFT JOIN is the following:

P_ID	FirstName	LastName	OrderName
1	John	Davis	NULL
2	Nick	Benson	Shirt
2	Nick	Benson	Jacket
3	Louise	Hodgson	Dress

SQL RIGHT JOIN

The SQL RIGHT JOIN returns all rows from the right table (table 2), even if there are no matches in the left table (table 1).

This means that if the ON clause matches 0 (zero) records in left table, the join will still return a row in the result, but with NULL in each column from left table.

SQL RIGHT JOIN

```
SELECT table1.column1, table2.column2...

FROM table1

RIGHT JOIN table2

ON table1.common_field = table2.common_field;
```

Example of SQL RIGHT JOIN

Imagine we have the following tables:

P_ID	FirstName	LastName
1	John	Davis
2	Nick	Benson
3	Louise	Hodgson

Persons Table

O_ID	OrderName	P_ID
1	Shirt	2
2	Jacket	2
3	Dress	3

Orders Table

Example of SQL RIGHT JOIN

```
SELECT Persons.P_ID, Persons.FirstName,
Persons.LastName, Orders.OrderName

FROM Persons

RIGHT JOIN Orders

ON Persons.P_ID=Orders.P_ID
```

Example of SQL RIGHT JOIN

The result of the RIGHT JOIN is the following:

P_ID	FirstName	LastName	OrderName
2	Nick	Benson	Shirt
2	Nick	Benson	Jacket
3	Louise	Hodgson	Dress

SQL FULL JOIN

The SQL FULL JOIN combines the results of both left and right outer joins.

The joined table will contain all records from both tables, and fill in NULLs for missing matches on either side.

Example of SQL FULL JOIN

Imagine we have the following tables:

Persons Table

P_ID	FirstName	LastName
1	John	Davis
2	Nick	Benson
3	Louise	Hodgson

Orders Table

O_ID	OrderName	P_ID
1	Shirt	2
2	Jacket	2
3	Dress	3

Example of SQL FULL JOIN

```
SELECT Persons.P_ID, Persons.FirstName,
Persons.LastName, Orders.OrderName

FROM Persons

FULL JOIN Orders

ON Persons.P_ID=Orders.P_ID
```

Example of SQL FULL JOIN

The result of the FULL JOIN is the following:

P_ID	FirstName	LastName	OrderName
1	John	Davis	NULL
2	Nick	Benson	Shirt
2	Nick	Benson	Jacket
3	Louise	Hodgson	Dress

SQL SELF JOIN

The SQL SELF JOIN is used to join a table to itself as if the table were two tables, temporarily renaming at least one table in the SQL statement.

```
SELECT a.column_name, b.column_name...
FROM table1 a, table1 b
WHERE a.common_field = b.common_field;
```

Example of SQL SELF JOIN

Imagine we have the following table:

Persons Table

P_ID	FirstName	LastName	Salary
1	John	Davis	4500
2	Nick	Benson	6000
3	Louise	Hodgson	2000

Example of SQL SELF JOIN

```
SELECT a.FirstName, a.LastName, a.Salary, b.Salary

FROM Persons a, Persons b

WHERE a.Salary < b.Salary;
```

Example of SQL SELF JOIN

FirstName	LastName	Salary	Salary
Louise	Hodgson	2000	4500
Louise	Hodgson	2000	6000
John	Davis	4500	6000

Example of SQL SELF JOIN

```
SELECT a.FirstName, a.LastName, a.Salary, b.Salary

FROM Persons a, Persons b

WHERE a.Salary = b.Salary;
```

Example of SQL SELF JOIN

FirstName	LastName	Salary	Salary
Louise	Hodgson	2000	2000
John	Davis	4500	4500
Nick	Benson	6000	6000

Example of SQL SELF JOIN

```
SELECT a.FirstName, a.LastName, a.Salary, b.Salary

FROM Persons a, Persons b

WHERE a.Salary > b.Salary;
```

Example of SQL SELF JOIN

FirstName	LastName	Salary	Salary
Nick	Benson	6000	4500
Nick	Benson	6000	2000
John	Davis	4500	2000

SQL CARTESIAN JOIN

The CARTESIAN JOIN or CROSS JOIN returns the Cartesian product of the sets of records from the two or more joined tables.

```
SELECT table1.column1, table2.column2...
FROM  table1, table2
```

Example of SQL CARTESIAN JOIN

Imagine we have the following tables:

P_ID	FirstName	LastName
1	John	Davis
2	Nick	Benson
3	Louise	Hodgson

Persons Table

O_ID	OrderName	P_ID
1	Shirt	2
2	Jacket	2
3	Dress	3

Orders Table

Example of SQL CARTESIAN JOIN

```
SELECT Persons.P_ID, Persons.LastName,
Persons.FirstName, Orders.OrderName

from Persons, Orders;
```

Example of SQL CARTESIAN JOIN

P_ID	FirstName	LastName	OrderName
3	Louise	Hodgson	Shirt
3	Louise	Hodgson	Jacket
3	Louise	Hodgson	Dress
1	John	Davis	Shirt
1	John	Davis	Jacket
1	John	Davis	Dress
2	Nick	Benson	Shirt
2	Nick	Benson	Jacket
2	Nick	Benson	Dress

Introduction to PL/SQL

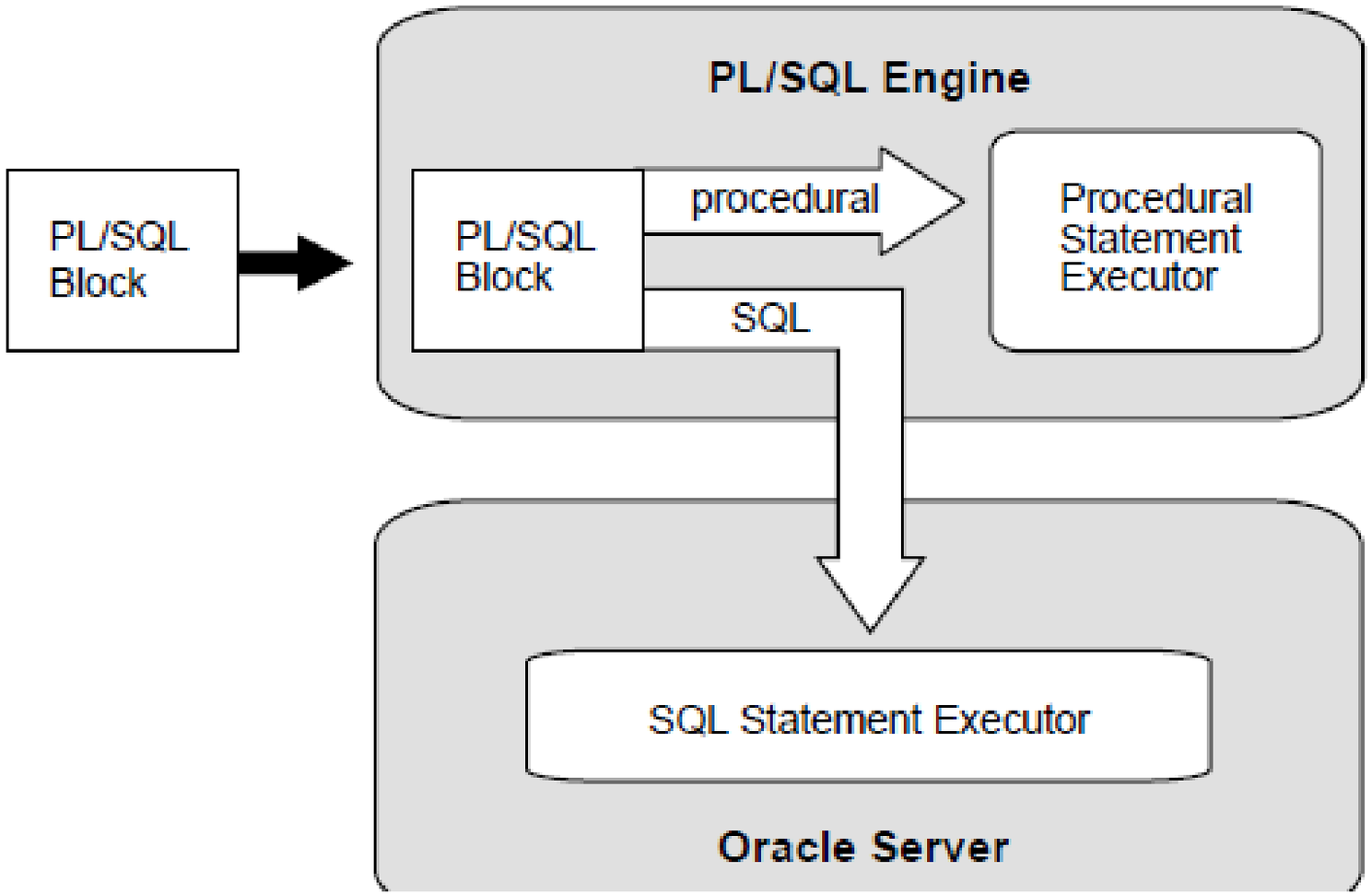
PL/SQL

- **P**(*rocedural*) **L**(*anguage*)***/*SQL**
- A combination of SQL along with the procedural features of programming languages.
- It was developed by Oracle Corporation in the early 90s to enhance the capabilities of SQL.

PL/SQL

- PL/SQL is a completely portable, high-performance transaction-processing language.
- PL/SQL provides an OS independent programming environment.
- PL/SQL can also directly be called from the command-line SQL*Plus interface.
- Direct call can also be made from external programming language calls to database.

PL/SQL Architecture



PL/SQL Advantages

- PL/SQL allows sending an entire block of statements to the database at one time.
- Provision of access to predefined SQL packages.
- Provision of support for Developing Web Applications and Server Pages.

PL/SQL Advantages

- Provides the ability to add middle tier business logic to client/server applications
- Provides Portability
- Improves performance of multi-query transactions
- Provides error handling

PL/SQL Program Structure

- PL/SQL is a **block-structured language**, meaning that PL/SQL programs are divided and written in logical blocks of code.

- Each block consists of three sub-parts:

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

2. **Executable Commands:** This section is enclosed between the keywords BEGIN and END and it is a **mandatory section**. It consists of the executable PL/SQL statements of the program. It should have at least one executable line of code

3. **Exception Handling:** This section starts with the keyword EXCEPTION. This section is again **optional** and contains exception(s) that handle errors in the program.

PL/SQL Program Structure

```
DECLARE
   <declarations section>
BEGIN
   <executable command(s)>
EXCEPTION
   <exception handling>
END;
```

The 'Hello World' Example:

```
DECLARE
   message  varchar2(20):= 'Hello, World!';
BEGIN
   dbms_output.put_line(message);
END;
```

PL/SQL Program Declarations Section

- The *declarations section* is the first section of the PL/SQL block.
- It contains definitions of PL/SQL identifiers such as variables, constants, cursors and so on.
- **Example**

```
DECLARE
        v_first_name VARCHAR2(35) ;
        v_last_name VARCHAR2(35) ;
        v_counter NUMBER := 0 ;
```

PL/SQL Program Executable Section

- This section contains executable statements that allow you to manipulate the variables that have been declared in the declaration section.

```
BEGIN

        SELECT first_name, last_name
                INTO v_first_name, v_last_name
                FROM student
                WHERE student_id = 123 ;
        DBMS_OUTPUT.PUT_LINE
        ('Student name :' || v_first_name || '  ' || v_last_name);

END;
```

PL/SQL Program Exception Handling Section

- This section contains statements that are executed when a runtime error occurs within a block.

- **Runtime errors occur while the program is running and cannot be detected by the PL/SQL compiler**.

```
EXCEPTION
    WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE
            (‘ There is no student with student id 123 ’);
END;
```

Fundamentals of PL/SQL

```
PROCEDURE print_date IS

  v_date VARCHAR2(30);

BEGIN

     SELECT TO_CHAR(SYSDATE,'Mon DD, YYYY')
          INTO v_date
          FROM DUAL;
     DBMS_OUTPUT.PUT_LINE(v_date);

END;
```

Fundamentals of PL/SQL

Comments

Single and Multi-line block comments

The single-line comment is initiated with two hyphens (--), which cannot be separated by a space or any other characters.

Multiline comments start with a slash-asterisk (/*) and end with an asterisk-slash (*/).

PL/SQL Variables

variable_name datatype = initial_value

variable_name is a valid identifier in PL/SQL, datatype must be a valid PL/SQL data type.

When you provide a size, scale or precision limit with the data type, it is called a constrained declaration.

Constrained declarations require less memory than unconstrained declarations.

***Example*:**

sales number(10, 2);

PL/SQL Variables

- Whenever you declare a variable, PL/SQL assigns it a default value of **NULL**.
- If you want to initialize a variable with a value other than the NULL value, you can do so during the declaration, using either of the following:
- **The DEFAULT keyword**
- **The assignment operator**

Example

```
counter binary_integer := 0;
greetings varchar2(20) DEFAULT 'Have a Good Day';
```

Scope of Variables PL/SQL

- PL/SQL allows the **nesting of Blocks**
- There are **two types of variable scope**:

1. **Local variables** - variables declared in an inner block and not accessible to outer blocks

2. **Global variables** - variables declared in the outermost block or a package

Scope of Variables PL/SQL

```
DECLARE
   -- Global variables
   num1 number := 95;
   num2 number := 85;
BEGIN
   dbms_output.put_line('Outer Variable num1: ' || num1);
   dbms_output.put_line('Outer Variable num2: ' || num2);
   DECLARE
      -- Local variables
      num1 number := 195;
      num2 number := 185;
   BEGIN
      dbms_output.put_line('Inner Variable num1: ' || num1);
      dbms_output.put_line('Inner Variable num2: ' || num2);
   END;
END;
```

PL/SQL Constants

- A constant holds a value that once declared, does not change in the program.
- A constant declaration specifies its name, data type, and value.
- A constant is declared using the **CONSTANT** keyword. It requires an initial value and does not allow that value to be changed.
- **Example**: salary_increase **CONSTANT** number (3) := 10;

PL/SQL Records

PL/SQL Records

- **A PL/SQL record is a data structure that can hold data items of different kinds.**

- **Records consist of different fields, similar to a row of a database table**.

- **A record can be visualized as a row of data**. It can contain all the contents of a row.

PL/SQL Records

The General Syntax to define a composite datatype is:

TYPE record_type_name IS RECORD
(first_col_name column_datatype,
second_col_name column_datatype, ...);

record_type_name – it is the name of the record you want to define.

first_col_name, second_col_name, etc.,- it is the names the fields/columns within the record.

column_datatype defines the scalar datatype of the fields.

PL/SQL Records

PL/SQL can handle the following types of records:

-Table-based

-Cursor-based records

-User-defined records

In this lecture, we will see the Table-based and User-defined records.

Table-based Records

The %ROWTYPE attribute enables a programmer to create table-based records

Example:

```
DECLARE
   customer_rec customers%rowtype;
BEGIN
   SELECT * into customer_rec
   FROM customers
   WHERE id = 5;

   dbms_output.put_line('Customer ID: ' || customer_rec.id);
   dbms_output.put_line('Customer Name: ' || customer_rec.name);
   dbms_output.put_line('Customer Address: ' || customer_rec.address);
   dbms_output.put_line('Customer Salary: ' || customer_rec.salary);
END;
```

User-defined Records

PL/SQL provides a user-defined record type that allows you to define different record structures

Example

```
DECLARE
TYPE books IS RECORD
(   title  varchar(50),
    author  varchar(50),
    subject varchar(100),
    book_id   number
);
```

PL/SQL Control Statements

PL/SQL Control Statements

Conditional selection statements (IF, CASE), which run different statements for different data values.

Loop statements (LOOP, FOR LOOP, WHILE LOOP), which run the same statements with a series of different data values.

The EXIT statement transfers control to the end of a loop. The CONTINUE statement exits the current iteration of a loop and transfers control to the next iteration. Both EXIT and CONTINUE have an optional WHEN clause, where you can specify a condition.

Sequential control statements (GOTO, NULL) , which are not crucial to PL/SQL programming.

PL/SQL Conditional Selection Statements

The IF statement either runs or skips a sequence of one or more statements, depending on a condition. The IF statement has these forms:

IF THEN
IF THEN ELSE
IF THEN ELSIF

The CASE statement chooses from a sequence of conditions, and runs the corresponding statement.

The CASE statement has these forms:

-***Simple, which evaluates a single expression and compares it to several potential values.***

-***Searched, which evaluates multiple conditions and chooses the first one that is true.***

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

```
IF condition THEN
  statements
ELSE
  else_statements
END IF;
```

```
IF condition_1 THEN
  statements_1
ELSIF condition_2 THEN
  statements_2
[ ELSIF condition_3 THEN
    statements_3
]...
[ ELSE
    else_statements
]
END IF;
```

CASE Statement

```
CASE selector

WHEN selector_value_1 THEN statements_1
WHEN selector_value_2 THEN statements_2
...
WHEN selector_value_n THEN statements_n

[ ELSE
  else_statements ]

END CASE;]
```

CASE Statement

```
DECLARE
  grade CHAR(1);
BEGIN
  grade := 'B';

  CASE grade

    WHEN 'A' THEN DBMS_OUTPUT.PUT_LINE('Excellent');
    WHEN 'B' THEN DBMS_OUTPUT.PUT_LINE('Very Good');
    WHEN 'C' THEN DBMS_OUTPUT.PUT_LINE('Good');
    WHEN 'D' THEN DBMS_OUTPUT.PUT_LINE('Fair');
    WHEN 'F' THEN DBMS_OUTPUT.PUT_LINE('Poor');
    ELSE DBMS_OUTPUT.PUT_LINE('No such grade’);

  END CASE;

END;
```

PL/SQL Loops

Loops

Basic LOOP Statement

FOR LOOP Statement

WHILE LOOP Statement

EXIT Statement

EXIT WHEN Statement

CONTINUE Statement

CONTINUE WHEN Statement

Basic LOOP Statement

```
DECLARE
  x NUMBER := 0;
BEGIN
  LOOP
    DBMS_OUTPUT.PUT_LINE ('Inside loop:  x = ' || TO_CHAR(x));
    x := x + 1;
    IF x > 3 THEN
      EXIT;
    END IF;
  END LOOP;
  -- After EXIT, control resumes here
  DBMS_OUTPUT.PUT_LINE(' After loop:  x = ' || TO_CHAR(x));
END;
```

Basic LOOP Statement

```
DECLARE
  x NUMBER := 0;
BEGIN
  LOOP
    DBMS_OUTPUT.PUT_LINE('Inside loop:  x = ' || TO_CHAR(x));
    x := x + 1;  -- prevents infinite loop
    EXIT WHEN x > 3;
  END LOOP;
  -- After EXIT statement, control resumes here
  DBMS_OUTPUT.PUT_LINE('After loop:  x = ' || TO_CHAR(x));
END;
```

Basic LOOP Statement

```
DECLARE
  x NUMBER := 0;
BEGIN
  LOOP -- After CONTINUE statement, control resumes here
    DBMS_OUTPUT.PUT_LINE ('Inside loop:  x = ' || TO_CHAR(x));
    x := x + 1;
    IF x < 3 THEN
      CONTINUE;
    END IF;
    DBMS_OUTPUT.PUT_LINE
      ('Inside loop, after CONTINUE:  x = ' || TO_CHAR(x));
    EXIT WHEN x = 5;
  END LOOP;

  DBMS_OUTPUT.PUT_LINE (' After loop:  x = ' || TO_CHAR(x));
END;
```

Basic LOOP Statement

```
DECLARE
  x NUMBER := 0;
BEGIN
  LOOP -- After CONTINUE statement, control resumes here
    DBMS_OUTPUT.PUT_LINE ('Inside loop:  x = ' || TO_CHAR(x));
    x := x + 1;
    CONTINUE WHEN x < 3;
    DBMS_OUTPUT.PUT_LINE
      ('Inside loop, after CONTINUE:  x = ' || TO_CHAR(x));
    EXIT WHEN x = 5;
  END LOOP;
  DBMS_OUTPUT.PUT_LINE (' After loop:  x = ' || TO_CHAR(x));
END;
```

FOR Loop

```
BEGIN

DBMS_OUTPUT.PUT_LINE ('lower_bound < upper_bound');

  FOR i IN 1..3 LOOP

    DBMS_OUTPUT.PUT_LINE (i);

  END LOOP;
```

WHILE Loop

```
WHILE monthly_value <= 4000

LOOP

   monthly_value := daily_value * 31;

END LOOP;
```

Some questions...

- What is a Procedure?
- What is a Function?

Two similar but different things

Modularization

- The process by which you break up large blocks of code into smaller pieces (modules) that can be called by other modules.

- With modularization, your code becomes:

  - More reusable

  - More manageable

  - More readable

  - More reliable

Modularization

Procedure

Function

Database trigger

Package

Procedures

- A **procedure** is a sequence of program instructions that perform a specific task, packaged as a unit.

- Procedures may be defined within programs, or separately in libraries that can be used by multiple programs.

- In different programming languages, a procedure may be called **a subroutine, a function, a routine, a method, or a callable unit**.

Procedures

Example from PASCAL

```
procedure name(argument(s): type1, argument(s): type 2, ... );
   < local declarations >
begin
   < procedure body >
end;
```

A procedure definition in Pascal consists of a header, local declarations and a body of the procedure. A procedure will also have following three parts:

1. **Declarative Part**
2. **Executable Part**
3. **Exception-handling**

Stored Procedures

- **A stored procedure is a subroutine available to applications that access a relational database management system (RDBMS). Such procedures are stored in the database data dictionary.**

- A stored procedure is also termed **proc, storp, sproc, StoPro, StoredProc, StoreProc, sp, or SP**.

- A stored procedure is a PL/SQL block that Oracle stores in the database and can be called **by name** from an application.

- **In Oracle, procedures and stored procedures are the same thing**.

Advantages of Stored Procedures

- Performance
- Productivity and Ease of Use
- Scalability
- Maintainability
- Interoperability
- Security
- Replication

Stored Procedures Creation

A procedure can be created at:

Schema Level

Inside a package

Inside a PL/SQL block

- A schema level procedure is a standalone procedure. It is created with the **CREATE PROCEDURE** statement. It is stored in the database and can be deleted with the **DROP PROCEDURE** statement.

- A procedure created inside a package is a packaged procedure. It is stored in the database and can be deleted only when the package is deleted with the DROP PACKAGE statement.

Stored Procedure Creation at Schema Level

```
CREATE OR REPLACE procedure_name
[(parameter_name [IN | OUT | IN OUT] type [, ...])]
{ IS | AS }

BEGIN
procedure_body

[EXCEPTION
   exception_section]
END procedure_name;
```

Example

```
CREATE OR REPLACE PROCEDURE greetings
AS
BEGIN
  dbms_output.put_line('Hello World!');
END;
```

Stored Procedure Creation inside a Package

```
CREATE OR REPLACE PACKAGE BODY emp_mgmt AS
  tot_emps NUMBER;
  tot_depts NUMBER;
PROCEDURE hire
  (last_name VARCHAR2, job_id VARCHAR2,
   manager_id NUMBER, salary NUMBER,
   commission_pct NUMBER, department_id NUMBER)
  RETURN NUMBER IS new_empno NUMBER;
BEGIN
  SELECT employees_seq.NEXTVAL
    INTO new_empno
    FROM DUAL;
  INSERT INTO employees
    VALUES (new_empno, 'First', 'Last','first.example@example.com',
          '(415)555-0100','18-JUN-02','IT_PROG',90000000,00,
          100,110);
    tot_emps := tot_emps + 1;
  RETURN(new_empno);
END;
END emp_mgmt;
```

Stored Procedure Creation

procedure-name specifies the name of the procedure.

[OR REPLACE] option allows modifying an existing procedure.

The optional parameter list contains name, mode and types of the parameters. **IN represents that value will be passed from outside and OUT represents that this parameter will be used to return a value outside of the procedure.**

procedure-body contains the executable part.

Stored Procedures Execution & Deletion

1. **Using the EXECUTE keyword**
2. **Calling the name of the procedure from a PL/SQL block**

The procedure 'greetings' can be called with the EXECUTE keyword:

```
EXECUTE greetings;
```

The procedure can also be called from another PL/SQL block:

```
BEGIN
   greetings;
END;
```

A standalone procedure is deleted with the DROP PROCEDURE statement. Syntax for deleting a procedure is:

```
DROP PROCEDURE procedure-name;
```

General Format of PL/SQL Procedure:

```
PROCEDURE apply_discount
 (company_id_in IN company.company_id%TYPE, discount_in IN NUMBER)
IS
 min_discount CONSTANT NUMBER:=.05;
 max_discount CONSTANT NUMBER:=.25;
 invalid_discount EXCEPTION;
BEGIN
 IF discount_in BETWEEN min_discount AND max_discount
  THEN
   UPDATE item
    SET item_amount:=item_amount*(1-discout_in);
   WHERE EXISTS (SELECT 'x' FROM order
     WHERE order.order_id=item.order_id
      AND order.company_id=company_id_in);
  IF SQL%ROWCOUNT = 0 THEN RAISE NO_DATA_FOUND; END IF;
 ELSE
  RAISE invalid_discount;
 END IF;
EXCEPTION
 WHEN invalid_discount
 THEN
  DBMS_OUTPUT.PUT_LINE('The specified discount is invalid.');

 WHEN NO_DATA_FOUND
 THEN
  DBMS_OUTPUT.PUT_LINE('No orders in the system for company:'||
     TO_CHAR(company_id_in));
END apply_discount;
```

Header

Declaration

Execution

Exception

The portion of the procedure definition that comes before the AS/IS keyword is called the procedure header or signature. The header provides all the information a programmer needs to call that procedure.

The body of the procedure is the code required to implement that procedure; it consists of the declaration, execution, and exception sections of the procedure.

You can append the name of the procedure directly after the END keyword when you
complete your procedure.

Passing Parameters

We can pass parameters to procedures in three ways.

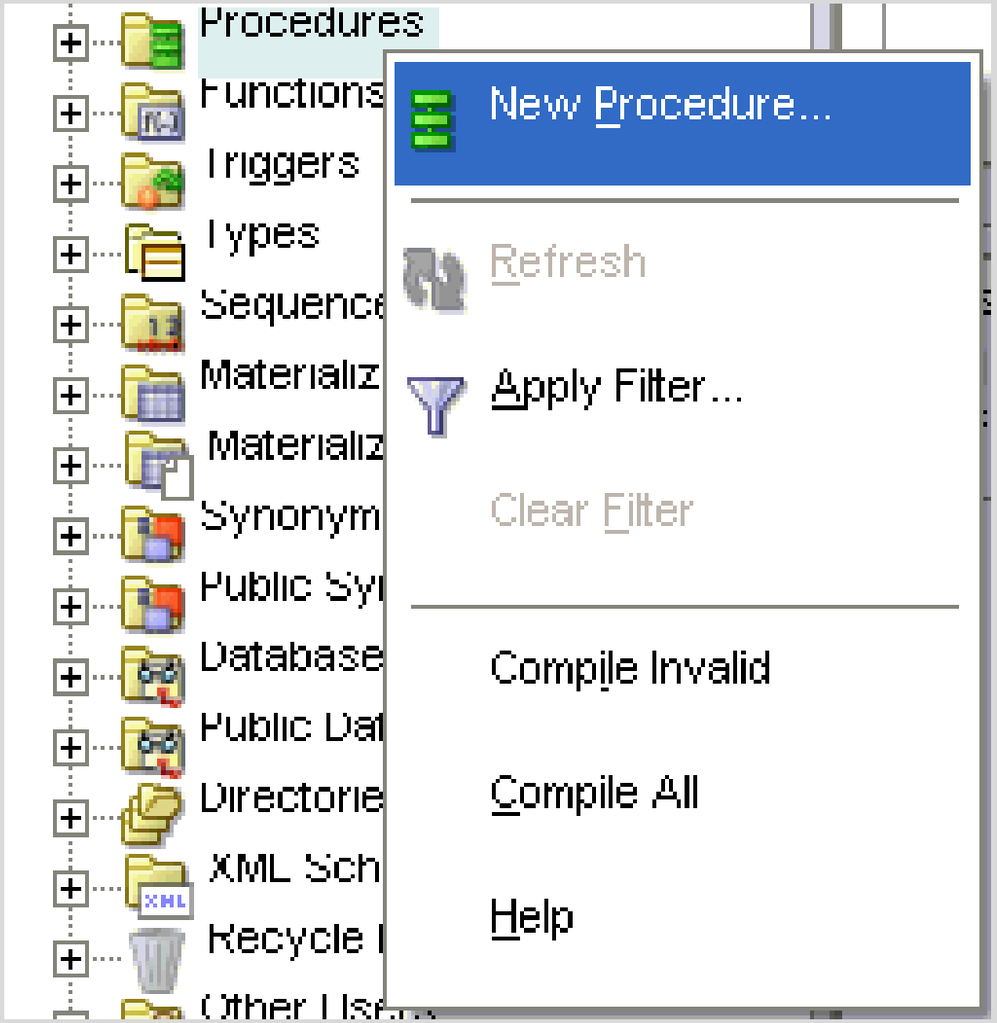
1) **IN-parameters**

2) **OUT-parameters**

3) **IN OUT-parameters**

Stored Procedures & Oracle Developer

- In the Connections navigation hierarchy, right-click Procedures.
- Select New Procedure.



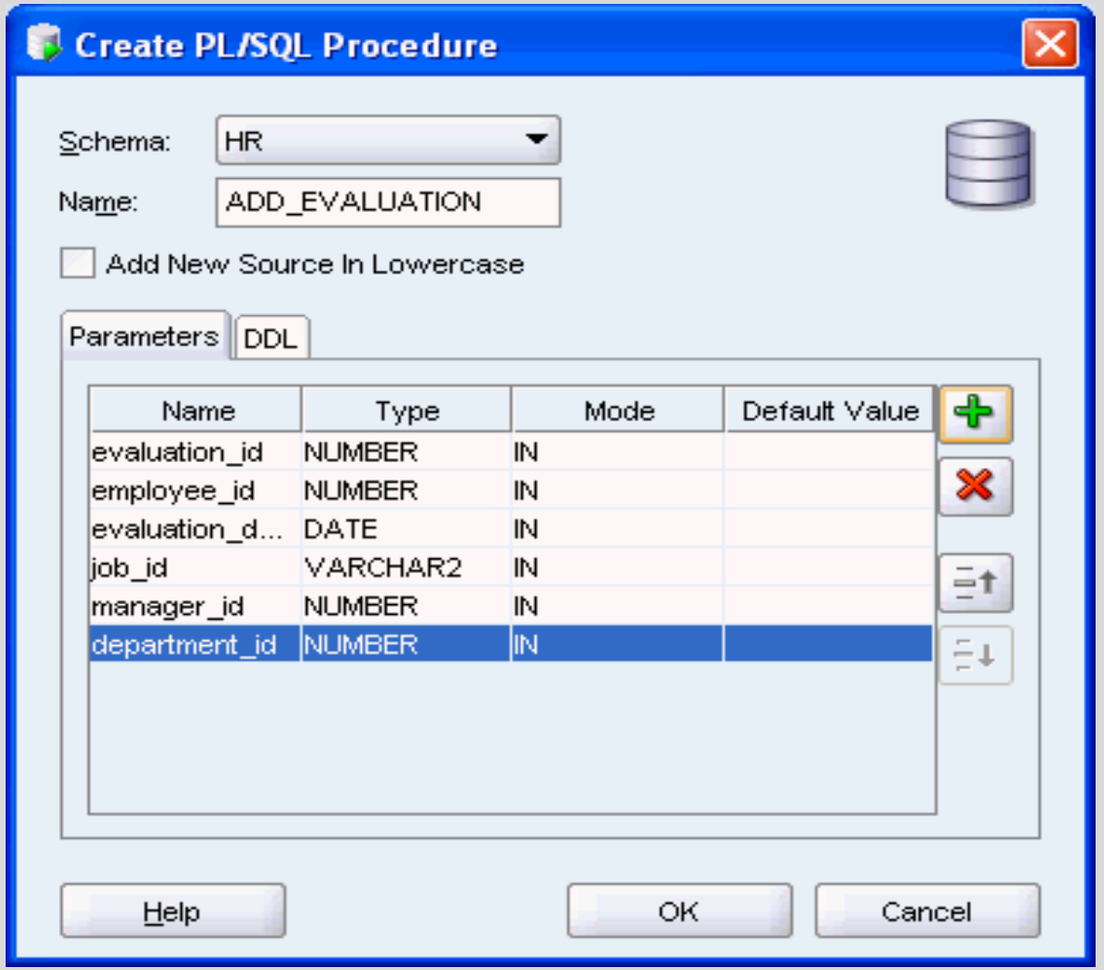
Stored Procedures & Oracle Developer

- In the New Procedure window, set the following parameters:

-**Ensure that you define a name for the Schema**.

-**Ensure that you set a name for the stored procedure**.

In the Parameters tab, click the Add Column icon ('plus' sign) and specify the first parameter of the procedure.



Functions

- A function is a named PL/SQL Block which is similar to a procedure. **The major difference between a procedure and a function is, a function must always return a value, but a procedure may not return a value**.

- The simplified syntax for the **CREATE OR REPLACE PROCEDURE** statement is as follows:

```
CREATE [OR REPLACE] FUNCTION function_name
[(parameter_name [IN | OUT | IN OUT] type [, ...])]

RETURN return_datatype {IS | AS}

BEGIN
  < function_body >
END [function_name];
```

Functions

- **function-name** specifies the name of the function.
- **[OR REPLACE]** option allows modifying an existing function.
- The optional **parameter list** contains name, mode and types of the parameters. IN represents that value will be passed from outside and OUT represents that this parameter will be used to return a value outside of the procedure.
- **The function must contain a return statement.**
- **RETURN** clause specifies that data type you are going to return from the function.
- **function-body** contains the executable part.

Examples

```
CREATE OR REPLACE FUNCTION minimum(v1 number, v2 number)
RETURN number IS
BEGIN
     IF v1 < v2 THEN
       RETURN v1;
       ELSE
         RETURN v2;
      END IF;
END;
```

```
CREATE FUNCTION get_bal(acc_no IN NUMBER)
RETURN NUMBER IS acc_bal NUMBER(11,2);
  BEGIN
    SELECT balance
    INTO acc_bal
    FROM accounts
    WHERE account_id = acc_no;
    RETURN (acc_bal);
   END;
```

Function with OUT parameter

```
CREATE OR REPLACE FUNCTION out_func (outparm OUT VARCHAR2)
RETURN VARCHAR2 IS

BEGIN
  outparm := 'out param';
  RETURN 'return param';
END out_func;
--------------------------------------------------------------------------------
set serveroutput on
--------------------------------------------------------------------------------
DECLARE
    retval VARCHAR2(20);
    outval VARCHAR2(20);
 BEGIN
        retval := out_func(outval);
        dbms_output.put_line(outval);
        dbms_output.put_line(retval);
END;
```

Function with IN OUT parameter

```
CREATE OR REPLACE FUNCTION inout_func (outparm IN OUT VARCHAR2)
RETURN VARCHAR2 IS
BEGIN
  outparm := 'Coming out';
  RETURN 'return param';
END inout_func;


-------------------------------------------------------
set serveroutput on
-------------------------------------------------------------------------------
DECLARE
  retval VARCHAR2(20);
  ioval  VARCHAR2(20) := 'Going in';
BEGIN
  dbms_output.put_line('In: ' || ioval);
  retval := inout_func(ioval);
  dbms_output.put_line('Out: ' || ioval);
  dbms_output.put_line('Return: ' || retval);
END;
```

The RETURN Clause

- **Return a string from a standalone function**

```
FUNCTION favorite_nickname (
name_in IN VARCHAR2)
RETURN VARCHAR2
IS
BEGIN
...
END;
```

- **Return a number (age of a pet) from an object type member function**

```
TYPE pet_t IS OBJECT (
tag_no INTEGER,
NAME VARCHAR2 (60),
breed VARCHAR2(100),
dob DATE,
MEMBER FUNCTION age RETURN NUMBER
)
```

The RETURN Clause

- **Return a record with the same structure as the books table**

```
PACKAGE book_info
IS
 FUNCTION onerow (isbn_in IN books.isbn%TYPE)
 RETURN books%ROWTYPE;
...
```

- **Return a cursor variable with the specified REF CURSOR type (based on a record type)**

```
PACKAGE book_info
IS
 TYPE overdue_rt IS RECORD (
 isbn books.isbn%TYPE,
 days_overdue PLS_INTEGER);
 TYPE overdue_rct IS REF CURSOR RETURN overdue_rt;
 FUNCTION overdue_info (username_in IN lib_users.username%TYPE)
 RETURN overdue_rct;
...
```

The RETURN Clause

The RETURN statement is generally associated with a function because it is required to RETURN a value from a function.

Interestingly, PL/SQL also allows you to use a RETURN statement in a procedure. The procedure version of the RETURN does not take an expression; it therefore cannot pass a value back to the calling program unit. The RETURN simply halts execution of the procedure and returns control to the calling code.

AVOID USING RETURN WITH PROCEDURES AS IT LEADS TO UNSTRUCTURED (SPAGHETTI) CODE!!!

General Format of PL/SQL Function

```
FUNCTION tot_sales
 (company_id_in IN company.company_id%TYPE,
  status_in IN order.status_code%TYPE:=NULL)
RETURN NUMBER
IS
 /*Internal upper-cased version of status code */
 status_int order.status_code%TYPE:=UPPER(status_in);

 /*Parameterized cursor returns total discounted sales. */
 CURSOR sales_cur (status_in IN status_code%TYPE) IS
  SELECT SUM (amount*discount)
   FROM item
  WHERE EXISTS (SELECT 'X' FROM order
     WHERE order.order_id=item.order_id
     AND company_id=company_id_in
     AND status_code LIKE status_in);

 /*Return value for function*/
 return_value NUMBER;
BEGIN
 OPEN sales_cur (status_int);
 FETCH sales_cur INTO return_value;
 IF sales_cur%NOTFOUND
 THEN
    CLOSE sales_cur;
    RETURN NULL;
 ELSE
    CLOSE sales_cur;
    RETURN return_value;
 END IF;
END tot_sales;
```

Header

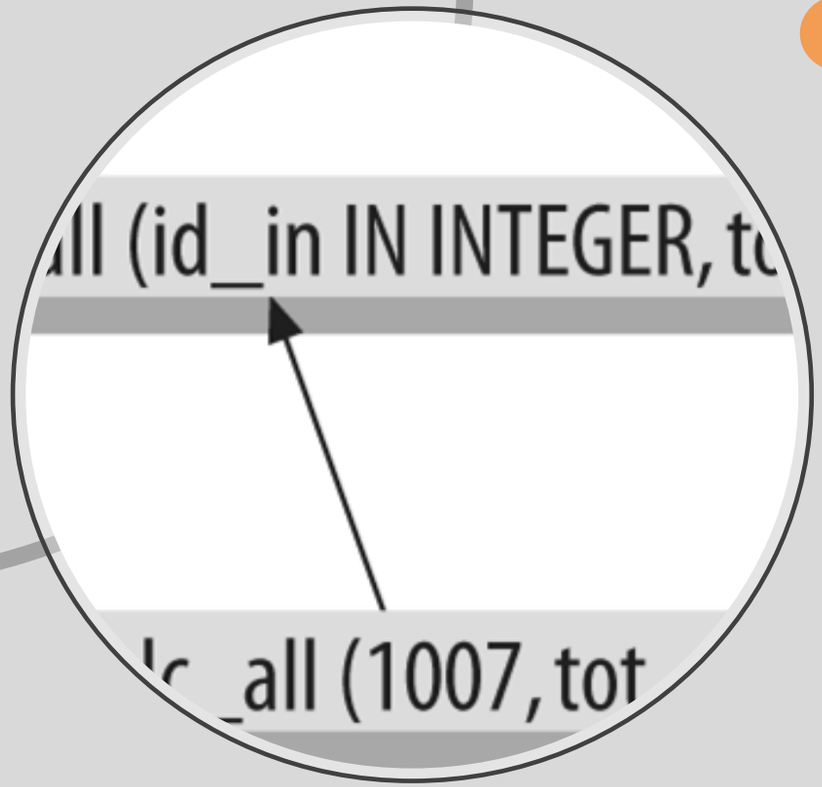
Declaration

Execution

The portion of the function definition that comes before the IS keyword is called the **function header or signature**. **The header provides all the information a programmer needs to call that function**.

The body of the function is the code required to implement the function. It consists of the declaration, execution, and exception sections of the function. Everything after the IS keyword in the function makes up that function's body.

A function must have at least one RETURN statement in its execution section. When a RETURN statement is processed, the function terminates immediately and returns control to the calling PL/SQL block. **The RETURN clause in the header of the function is different from the RETURN statement in the execution section of the body**.



Association of Actual with Formal Parameters

formal parameter (the name of the parameter)
actual parameter (the value of the parameter)

PL/SQL offers two ways to make the association:

Positional notation

Associates the actual parameter implicitly (by position) with the formal parameter

Named notation

Associates the actual parameter explicitly with the formal parameter, using the formal parameter's name and the **=>** symbol

The general syntax for named notation is:

formal_parameter_name => argument_value

Calling a Function

A function can be executed in the following ways.

1) Since a function returns a value, we can assign it to a variable

```
employee_name :=  employer_details_func;
```

If ‘employee_name’ is of datatype varchar we can store the name of the employee by assigning the return type of the function to it.

2) As a part of a SELECT statement

```
SELECT employer_details_func FROM dual;
```

3) In PL/SQL Statements

```
dbms_output.put_line(employer_details_func);
```

This line displays the value returned by the function.

Calling a Function

4) Assign the default value of a variable with a function call

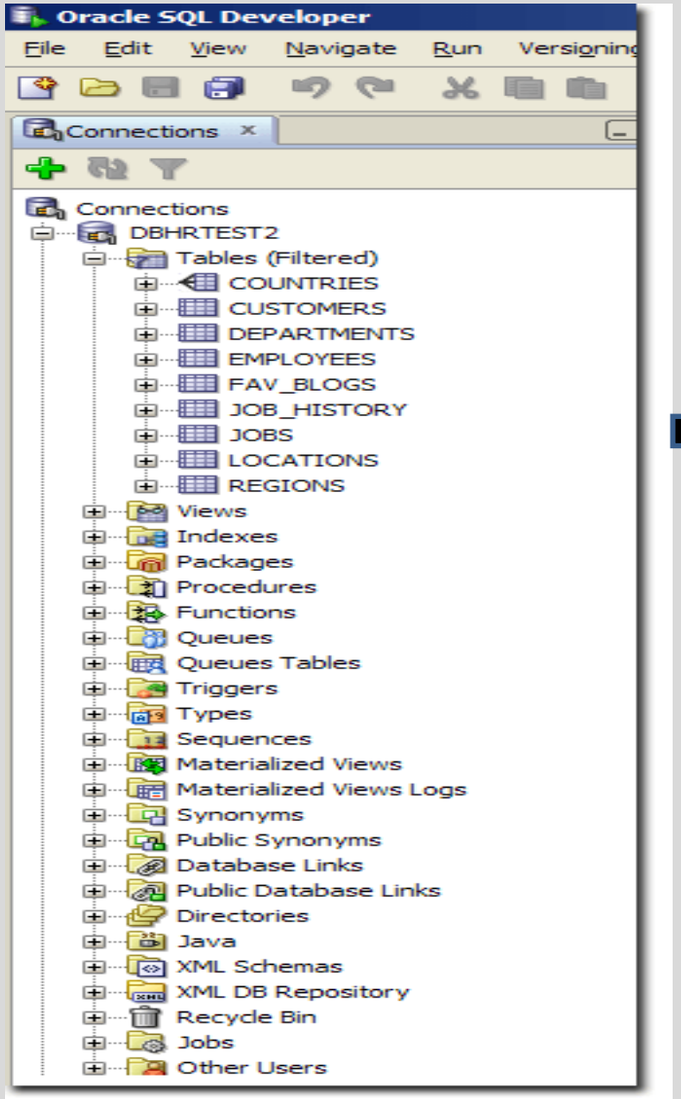
```
DECLARE
v_nickname VARCHAR2(100) := favorite_nickname ('Steven');
```

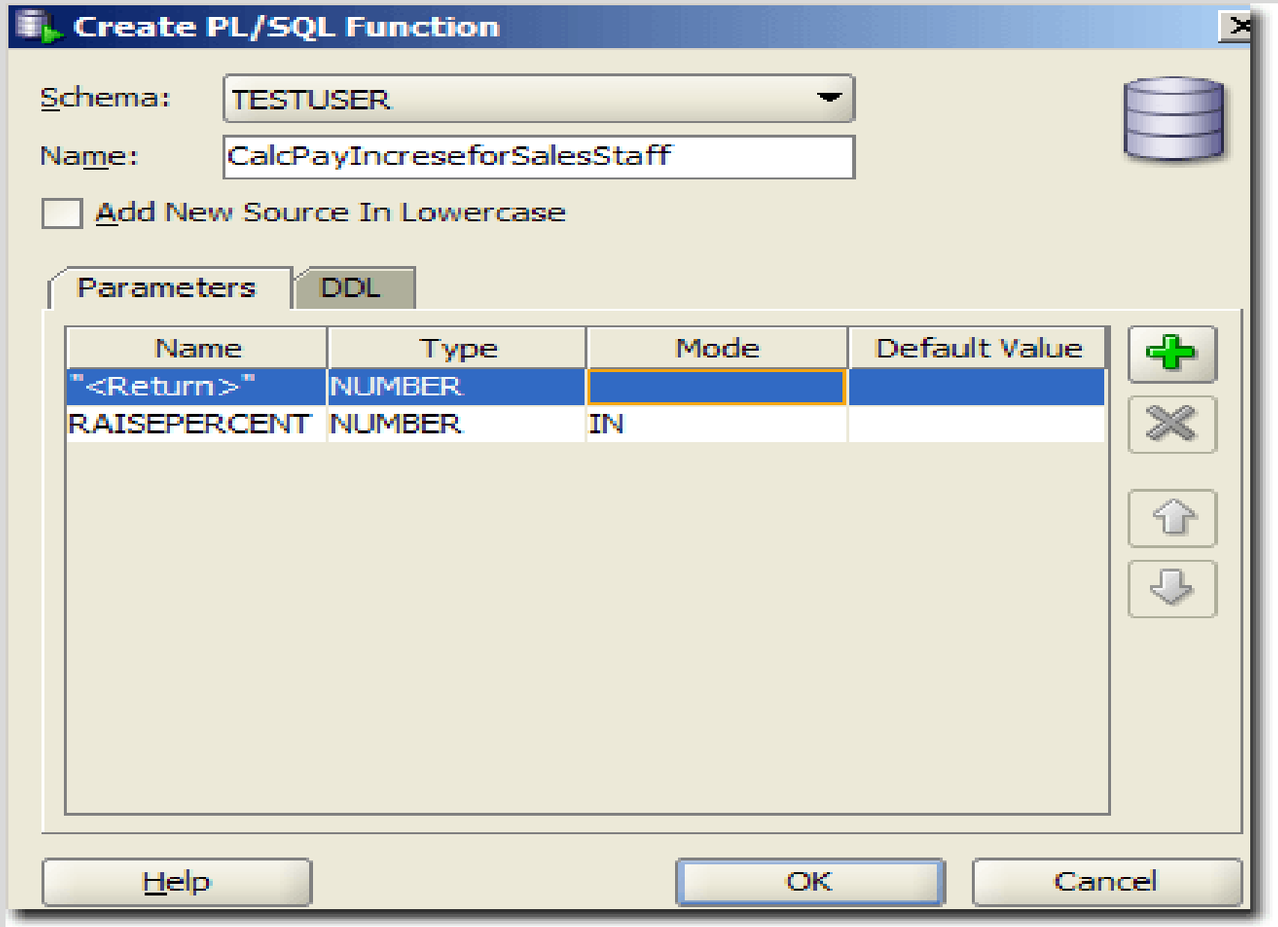
5) Call a user-defined PL/SQL function from within a query

```
DECLARE
l_name employees.last_name%TYPE;
BEGIN
SELECT last_name INTO l_name
FROM employees
WHERE employee_id = hr_info_pkg.employee_of_the_month ('FEBRUARY');
```

Functions and Oracle SQL Developer

- To get started, right click on Functions in the connection explorer and choose New Function from the drop down menu. Doing so should display the Create PL/SQL Function window.





Functions vs. Stored Procedures

- A Function must return a value but in a Stored Procedure this is optional
- Functions can be called from a Procedure whereas Procedures cannot be called from a Function
- Procedures cannot be utilized in a SELECT statement whereas a Function can be embedded in a SELECT statement
- Exception can be handled by the try-catch block in a Procedure whereas the try-catch block cannot be used in a Function
- Procedures allow SELECT as well as DML(INSERT/UPDATE/DELETE) statements in it whereas functions allow only SELECT statement in them

Packages

- **PL/SQL packages are schema objects that group logically related PL/SQL types, variables and subprograms**.
- A package is compiled and stored in the database
- A package has two mandatory parts:
- **Package specification**
- **Package body or definition**

Why do we use Packages?

- **Modularity**
- **Easier Application Design**
- **Information Hiding**
- **Better Performance**

Package Specification

- The specification is the **interface to the package**. **It just DECLARES the types, variables, constants, exceptions, cursors, and subprograms that can be referenced from outside the package.** In other words, it contains all information about the content of the package, but excludes the code for the subprograms.

Package Body

- The package body has the codes for various methods declared in the package specification and other private declarations, which are hidden from code outside the package.

- The **CREATE PACKAGE BODY** Statement is used for creating the package body.

Package Specification

The following code snippet shows a package specification having a single procedure. You can have many global variables defined and multiple procedures or functions inside a package.

```
CREATE PACKAGE cust_sal AS
   PROCEDURE find_sal(c_id customers.id%type);
END cust_sal;
```

Package Body

The following code snippet shows the package body declaration for the cust_sal package created above. We assume that we already have CUSTOMERS table created in our database.

```
CREATE OR REPLACE PACKAGE BODY cust_sal AS
  PROCEDURE find_sal(c_id customers.id%TYPE) IS
  c_sal customers.salary%TYPE;
  BEGIN
    SELECT salary INTO c_sal
    FROM customers
    WHERE id = c_id;
    dbms_output.put_line('Salary: '|| c_sal);
  END find_sal;
END cust_sal;
```

Using the Package Elements

The package elements (variables, procedures or functions) are accessed with the following syntax: **package_name.element_name;**

Example

Suppose we have the following table:

SQL> Select * from customers;

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  3000.00 |
|  2 | Khilan   |  25 | Delhi     |  3000.00 |
|  3 | kaushik  |  23 | Kota      |  3000.00 |
|  4 | Chaitali |  25 | Mumbai    |  7500.00 |
|  5 | Hardik   |  27 | Bhopal    |  9500.00 |
|  6 | Komal    |  22 | MP        |  5500.00 |
```

We specify the package body:

```
CREATE OR REPLACE PACKAGE BODY c_package AS
  PROCEDURE addCustomer(c_id  customers.id%type,
    c_name customers.name%type,
    c_age  customers.age%type,
    c_addr  customers.address%type,
    c_sal   customers.salary%type)
  IS
  BEGIN
    INSERT INTO customers (id,name,age,address,salary)
      VALUES(c_id, c_name, c_age, c_addr, c_sal);
  END addCustomer;

  PROCEDURE delCustomer(c_id   customers.id%type) IS
  BEGIN
     DELETE FROM customers
      WHERE id = c_id;
  END delCustomer;

  PROCEDURE listCustomer IS
  CURSOR c_customers is
    SELECT  name FROM customers;
  TYPE c_list is TABLE OF customers.name%type;
  name_list c_list := c_list();
  counter integer :=0;
  BEGIN
    FOR n IN c_customers LOOP
    counter := counter +1;
    name_list.extend;
    name_list(counter)  := n.name;
    dbms_output.put_line('Customer(' ||counter|| ')'||name_list(counter));
    END LOOP;
  END listCustomer;
END c_package;
```

USING THE PACKAGE:

```
DECLARE
  code customers.id%type:= 8;
BEGIN
     c_package.addcustomer(7, 'Rajnish', 25, 'Chennai',
3500);
     c_package.addcustomer(8, 'Subham', 32, 'Delhi',
7500);
     c_package.listcustomer;
     c_package.delcustomer(code);
     c_package.listcustomer;
END;
```

TRIGGERS

Triggers

– A trigger is an SQL statement (or a group of statements) that is automatically executed by Oracle in response to any of the following (triggering) events:

*DML statements

*DDL statements

*Database events, such as logon/logoff, errors, or start-up/shutdown

Triggers

-There are different types of triggers (**Row & Statement Triggers / BEFORE, AFTER, INSTEAD OF Triggers / Triggers on System Events and User Events**)

When creating a trigger, three pieces of information need to be specified:

1. **The unique trigger name**
2. **The table to which the trigger refers to**
3. **The action that the trigger should respond to**

Triggers are created using the **CREATE TRIGGER** statement

Parts of a Trigger

A trigger has three basic parts: **1. A triggering event or statement, 2. A trigger restriction (optional), 3. A trigger action**

```
CREATE OR REPLACE TRIGGER display_salary_changes
BEFORE DELETE OR INSERT OR UPDATE ON customers
FOR EACH ROW          (1)
WHEN (NEW.ID > 0)     (2)
DECLARE
   sal_diff number;
          BEGIN
          sal_diff := :NEW.salary  - :OLD.salary;
          dbms_output.put_line('Old salary: ' || :OLD.salary);
          (3)
          dbms_output.put_line('New salary: ' || :NEW.salary);
          dbms_output.put_line('Salary difference: ' || sal_diff);
          END;
```

Row/Statement Triggers

-A **row trigger** is executed each time the table is affected by the triggering statement. For example, if an INSERT or an UPDATE statement inserts or updates multiple rows of a table respectively, a row trigger is fired once **for each row affected by the INSERT or the UPDATE statement**. If a triggering statement affects no rows, a row trigger is not run.

-A **statement trigger** is executed regardless of the number of rows in the table that the triggering statement affects. A statement trigger is fired once on behalf of the triggering statement. For example, if a DELETE statement deletes several rows from a table, a statement-level DELETE trigger is fired only once.

Statement triggers are used to create audit records for security purposes.

BEFORE/AFTER/INSTEAD OF Triggers

BEFORE triggers run the trigger action before the triggering statement is run.

AFTER triggers run the trigger action after the triggering statement is run and in order to provoke another event to happen.

-*BEFORE and AFTER triggers fired by DML statements can be defined only on tables, and not on views. Triggers on the base tables of a view are fired if an INSERT, UPDATE, or DELETE statement is issued against the view*.

-**INSTEAD OF triggers override the standard actions of the triggering statement: an INSERT, UPDATE, or DELETE**. INSTEAD OF triggers can be defined on either tables or views.

System and User Events Triggers

-Triggers can be applied to System and User Events.

Examples of System Events are Database start-up and shutdown and Server error message events. User Events are User logon and logoff.

```
CREATE TRIGGER register_shutdown
  ON DATABASE
  SHUTDOWN
    BEGIN
    ...
    DBMS_AQ.ENQUEUE(...);
    ...
    END;
```

DBMS_AQ is a package used by Oracle and provides an interface to the Oracle Streams Advanced Queuing.

Syntax of a Trigger

- A single trigger can be associated with multiple events. If we want for example, a trigger to be executed for both the INSERT and UPDATE operations, we can define it as **AFTER INSERT, UPDATE**.
- Most triggers are AFTER triggers (they are executed after an event occurs). However, there are other types of trigger called BEFORE, INSTEAD OF.

- **Syntax of a Trigger**

```
CREATE TRIGGER <trigger_name> on <table | view>
AFTER | BEFORE | INSTEAD OF
INSERT | UPDATE | DELETE
AS
BEGIN
<trigger_code>
END
```

Manipulating Triggers

- **Drop a trigger**

  DROP TRIGGER <name_of_trigger>;

- **Disable/Enable a trigger**

  ALTER TRIGGER <name_of_trigger> DISABLE/ENABLE;

- **To check whether there are triggers in our database:**

  select * from USER_TRIGGERS;

- **To modify an existing trigger, we use the**

  ALTER TRIGGER statement

Triggers vs. Constraints

A trigger is a program unit that can be executed based on a DML operation or a DDL operation or a DATABASE event like logon and logoff. With triggers you specify how to handle data (in inserts, updates etc.).

Triggers are reactive: they undo the damage. Triggers are useful for database security.

A constraint is merely a rule that control the data that can be stored/allowed in database objects like tables and views.

Constraints are proactive: They prevent unwanted actions from happening. Constraints are useful for data consistency.

PL/SQL CURSORS

PL/SQL Cursors

- Oracle creates a memory area, known as **context area**, for processing an SQL statement
- A cursor is **a pointer to this context area**
- The context area includes all the information required to process a statement
- Cursors allow you to fetch and process rows returned by a SELECT statement, one row at a time.
- Two types of Cursors: **Implicit** and **Explicit**

PL/SQL Cursors Creation

- **CURSOR WITHOUT PARAMETERS (SIMPLEST)**
- **CURSOR WITH PARAMETERS**
- **CURSOR WITH RETURN CLAUSE**

Cursors without Parameters

```
CURSOR cursor_name
IS
  SELECT_statement;
```

Example

```
CURSOR c1
IS
  SELECT course_number
  FROM courses_tbl
  WHERE course_name = name_in;
```

Cursors with Parameters

```
CURSOR cursor_name (parameter_list)
IS
  SELECT_statement;
```

Example

```
CURSOR c2 (subject_id_in IN varchar2)
IS
  SELECT course_number
  FROM courses_tbl
  WHERE subject_id = subject_id_in;
```

Cursors with RETURN Clause

```
CURSOR cursor_name
RETURN field%ROWTYPE
IS
   SELECT_statement;
```

Example

```
CURSOR c3
RETURN courses_tbl%ROWTYPE
IS
   SELECT *
   FROM courses_tbl
   WHERE subject = 'Mathematics';
```

Implicit Cursors

- **Implicit cursors are automatically created by Oracle whenever an SQL statement is executed and ONLY if there is no explicit cursor already created for the statement.**
- Programmers cannot control the implicit cursors and the information in it.
- Whenever a DML statement (INSERT, UPDATE and DELETE) is issued, an implicit cursor is associated with this statement.
- In PL/SQL, the implicit cursor can be referred as the **SQL cursor**.

Implicit Cursors

- The implicit cursor attributes are **%FOUND, %ISOPEN %NOTFOUND, and %ROWCOUNT**

- **%FOUND**: Returns TRUE if an INSERT, UPDATE, or DELETE statement affected one or more rows or a SELECT INTO statement returned one or more rows. Otherwise, it returns FALSE.

- **%NOTFOUND**: The logical opposite of %FOUND. It returns TRUE if an INSERT, UPDATE, or DELETE statement affected no rows, or a SELECT INTO statement returned no rows. Otherwise, it returns FALSE.

- **%ISOPEN**: Always returns FALSE for implicit cursors, because Oracle closes the SQL cursor automatically after executing its associated SQL statement.

- **%ROWCOUNT**: Returns the number of rows affected by an INSERT, UPDATE, or DELETE statement, or returned by a SELECT INTO statement.

Explicit Cursors

- Explicit cursors are programmer-defined cursors for gaining more control over the context area. An explicit cursor should be defined in the declaration section of the PL/SQL Block.

CURSOR cursor_name IS select_statement;

- **Four Steps**:

1. DECLARING the cursor for initializing in the memory
eg. CURSOR c_customers IS
SELECT id, name, address FROM customers;

2. OPENING the cursor for allocating memory
eg. OPEN c_customers;

3. FETCHING the cursor for retrieving data
eg. FETCH c_customers INTO c_id, c_name, c_addr;

4. CLOSING the cursor to release allocated memory
eg. CLOSE c_customers;

```
SET SERVEROUTPUT ON
DECLARE

--DECLARE VARIABLES
v_product_id products.product_id%TYPE;
v_name products.name%TYPE;
v_price products.price%TYPE;

--DECLARE THE CURSOR
CURSOR cv_product_cursor IS
SELECT product_id, name, price
FROM products
ORDER BY product_id;
BEGIN

--OPEN THE CURSOR
OPEN cv_product_cursor;
LOOP
```

```
--FETCH THE ROWS FROM THE CURSOR
FETCH cv_product_cursor
INTO v_product_id, v_name, v_price;

--EXIT THE LOOP WHEN THERE ARE NO MORE ROWS
EXIT WHEN cv_product_cursor%NOTFOUND;

--USE DBMS_OUTPUT.PUT_LINE() TO DISPLAY THE VARIABLES
DBMS_OUTPUT.PUT_LINE(
'v_product_id = ' || v_product_id || ', v_name = ' || v_name || ', v_price = ' || v_price
);
END LOOP;

--CLOSE THE CURSOR
CLOSE cv_product_cursor;
END;
```

Using Cursors with FOR Loops

The syntax for the CURSOR FOR LOOP in Oracle/PLSQL is:

FOR *record_index* in *cursor_name*
LOOP
{...statements...}
END LOOP;

Parameters or Arguments

record_index

The index of the record.

cursor_name

The name of the cursor that you wish to fetch records from.

statements

The statements of code to execute each pass through the CURSOR FOR LOOP.

Example

```
BEGIN
  FOR employee_rec IN (
        SELECT *
        FROM employees
        WHERE department_id = 10)
  LOOP
    DBMS_OUTPUT.put_line (
      employee_rec.last_name);
  END LOOP;
END;
-------------------------------------------------------------------------------------------------------------------------
DECLARE
  CURSOR employees_in_10_cur
  IS
      SELECT *
      FROM employees
      WHERE department_id = 10;
BEGIN
     FOR employee_rec  IN employees_in_10_cur
     LOOP
     DBMS_OUTPUT.put_line (employee_rec.last_name);
     END LOOP;
END;
```

Cursor within a Cursor (Nested Cursors)

```
CREATE OR REPLACE PROCEDURE MULTIPLE_CURSORS_PROC is
 v_owner varchar2(40);
 v_table_name varchar2(40);
 v_column_name varchar2(100);

  /* First cursor */
    CURSOR get_tables IS
    SELECT DISTINCT tbl.owner, tbl.table_name
    FROM all_tables tbl WHERE tbl.owner = 'SYSTEM';

  /* Second cursor */
    CURSOR get_columns IS
    SELECT DISTINCT col.column_name
    FROM all_tab_columns col WHERE col.owner = v_owner
    AND col.table_name = v_table_name;
```

Cursor within a Cursor (Nested Cursors)

```
BEGIN
   -- Open first cursor
      OPEN get_tables;
   LOOP
     FETCH get_tables INTO v_owner, v_table_name;
     -- Open second cursor
        OPEN get_columns;
     LOOP
        FETCH get_columns INTO v_column_name;
     END LOOP;
       CLOSE get_columns;
   END LOOP;
       CLOSE get_tables;
EXCEPTION
   WHEN OTHERS THEN
raise_application_error(-20001,'An error was encountered - '||SQLCODE||' -ERROR- '||SQLERRM);
end MULTIPLE_CURSORS_PROC;
```

PL/SQL REF-CURSORS

Ref Cursors

- The REF CURSOR is a data type in Oracle.
- **Pointer to a Cursor, not the Cursor itself.**
- A REF CURSOR is not updatable. The result set represented by the REF CURSOR is **read-only**. You cannot update the database by using a REF CURSOR.
- You create and return a REF CURSOR inside a PL/SQL code block.

Ref Cursors

- A ref cursor in Oracle PL/SQL is much like an ordinary PL/SQL cursor in that **it acts as a pointer to the result set of the cursor with which it is associated.**

- **A ref cursor can be assigned to different result sets whereas a cursor is always associated with the same result set.**

- **The real purpose of ref cursors is to be able to share cursors and result sets between the user and the Oracle server or between different subroutines**.

Ref Cursors

- **Four Steps (as in the 'normal' cursors)**:

  1. DECLARE THE REF CURSOR

  2. OPEN THE REF CURSOR

  3. FETCH THE REF CURSOR

  4. CLOSE THE REF CURSOR

Ref Cursors

There are two steps required to create a cursor variable. First define a ref cursor TYPE; then declare cursor variable(s) of that type.

Define a REF Cursor TYPE:

```
TYPE ref_type_name IS REF CURSOR

  [RETURN {cursor_name%ROWTYPE
      |ref_cursor_name%ROWTYPE
      |record_name%TYPE
      |record_type_name
      |table_name%ROWTYPE} ];
```

The RETURN clause is optional, when included it causes the cursor variable to be strongly typed.

It is a good practice to use strongly typed cursor variable types - this ensures that columns returned by a query will match the return type of the cursor.

Cursor_variable_declaration: `ref_cursor_name  ref_type_name;`

Ref Cursors Categories

1. Strong Ref Cursor

Ref Cursors **which has a return type** is classified as **Strong Ref Cursor.**

Example

```
Declare
TYPE empcurtyp IS REF CURSOR
RETURN emp%ROWTYPE;
…..
End;
```

--Here empcurtyp is a Strong Ref Cursor

Ref Cursors Categories

2. Weak Ref Cursor

Ref Cursors **which has no return type** is classified as **Weak Ref Cursor.**

Example

```
Declare
TYPE empcurtyp IS REF CURSOR;
…..
End;
```

--Here empcurtyp is a Weak Ref Cursor

Ref Cursors Categories

3. System Ref Cursor

This is a system defined Ref Cursor. This also considered weak. System Ref Cursor does not need to be declared explicitly.

Example

```
Declare
empcurtyp SYS_REFCURSOR;
…..
End;
```

EXCEPTIONS

Exceptions

- A warning or error condition is called an **exception**.
- Exceptions can be internally defined (by the run-time system) *OR* user-defined.
- Examples of internally defined exceptions include division by zero and out of memory.

Exceptions

- You can define exceptions of your own in the declarative part of any PL/SQL block, subprogram, or package.

- PL/SQL supports programmers to catch such conditions using the **EXCEPTION** block in the program and an appropriate action is taken against the error condition.

Exceptions

- To handle raised exceptions, you write separate routines called ***exception handlers***.
- When an error occurs, an exception is raised.
- In this case, normal execution stops and control transfers to the exception-handling part of your PL/SQL block or subprogram.
- **Internal exceptions are raised implicitly (automatically) by the run-time system.**
- **User-defined exceptions must be raised explicitly by RAISE statements, which can also raise predefined exceptions.**

Exceptions

```
DECLARE
  <declarations section>
BEGIN
  <executable command(s)>
EXCEPTION
  <exception handling goes here >
  WHEN exception1 THEN
     exception1-handling-statements
  WHEN exception2  THEN
    exception2-handling-statements
  WHEN exception3 THEN
    exception3-handling-statements
  ........
  WHEN others THEN
    exception3-handling-statements
END;
```

Example

```
DECLARE
   c_id customers.id%type := 8;
   c_name  customers.name%type;
   c_addr customers.address%type;

BEGIN
   SELECT  name, address INTO  c_name, c_addr
   FROM customers
   WHERE id = c_id;

   DBMS_OUTPUT.PUT_LINE ('Name: '||  c_name);
   DBMS_OUTPUT.PUT_LINE ('Address: ' || c_addr);

EXCEPTION
   WHEN no_data_found THEN
      dbms_output.put_line('No such customer!');
   WHEN others THEN
      dbms_output.put_line('Error!');
END;
```

Raising Exceptions

- **Exceptions are raised by the database server automatically whenever there is any internal database error.**

- **Exceptions can be raised explicitly by the programmer by using the command RAISE**

```
DECLARE
   exception_name EXCEPTION;
BEGIN
   IF condition THEN
      RAISE exception_name;
   END IF;
EXCEPTION
   WHEN exception_name THEN
   statement;
END;
```

User-defined Exceptions

- PL/SQL allows you to define your own exceptions according to the need of your program.
- **A user-defined exception must be declared and then raised explicitly, using either a RAISE statement or the procedure DBMS_STANDARD.RAISE_APPLICATION_ERROR**.
- The syntax for **declaring an exception** is:

```
DECLARE
    my-exception EXCEPTION;
```

Example

```
DECLARE
   c_id customers.id%type := &cc_id;
   c_name  customers.name%type;
   c_addr customers.address%type;
   -- user defined exception
   ex_invalid_id  EXCEPTION;
BEGIN
   IF c_id <= 0 THEN
      RAISE ex_invalid_id;
   ELSE
      SELECT  name, address INTO  c_name, c_addr
      FROM customers
      WHERE id = c_id;
      DBMS_OUTPUT.PUT_LINE ('Name: '||  c_name);
      DBMS_OUTPUT.PUT_LINE ('Address: ' || c_addr);
   END IF;
EXCEPTION
   WHEN ex_invalid_id THEN
      dbms_output.put_line('ID must be greater than zero!');
   WHEN no_data_found THEN
      dbms_output.put_line('No such customer!');
   WHEN others THEN
      dbms_output.put_line('Error!');
END;
```

Pre-defined Exceptions

- PL/SQL provides many **pre-defined exceptions**, which are executed when any database rule is violated by a program.

- **Examples**

NO_DATA_FOUND: It is raised when a SELECT INTO statement returns no rows

LOGIN_DENIED: It is raised when s program attempts to log on to the database with an invalid username or password

ROWTYPE_MISMATCH: It is raised when a cursor fetches value in a variable having incompatible data type

ZERO_DIVIDE: It is raised when an attempt is made to divide a number by zero

End of Booklet