<https://www.guru99.com/complex-data-types-pl-sql.html>

**Difference between File system & DBMS**

**File System**

1. File system is a collection of data. Any management with the file system, user has to

write the procedures

2. File system gives the details of the data representation and Storage of data.

3. In File system storing and retrieving of data cannot be done efficiently.

4. Concurrent access to the data in the file system has many problems like

a. Reading the file while other deleting some information, updating some information

5. File system doesn’t provide crash recovery mechanism.

Eg. While we are entering some data into the file if System crashes then content of the

file is lost.

6. Protecting a file under file system is very difficult.

**DBMS**

1. DBMS is a collection of data and user is not required to write the procedures for

managing the database.

2. DBMS provides an abstract view of data that hides the details.

3. DBMS is efficient to use since there are wide varieties of sophisticated techniques to

store and retrieve the data.

4. DBMS takes care of Concurrent access using some form of locking.

5. DBMS has crash recovery mechanism, DBMS protects user from the effects of system

failures.

6. DBMS has a good protection mechanism.

**DBMS = Database Management System**

**RDBMS = Relational Database Management System**

**Advantages of DBMS.**

Due to its centralized nature, the database system can overcome the disadvantagesof the file system-based system

1. **Data independency**:

Application program should not be exposed to details of data representation and storage

DBMS provides the abstract view that hides these details.

2. **Efficient data access**.:

DBMS utilizes a variety of sophisticated techniques to store and retrieve dataefficiently.

3. **Data integrity and security**:

Data is accessed through DBMS, it can enforce integrity constraints.

E.g.: Inserting salary information for an employee.

4. **Data Administration**:

When users share data, centralizing the data is an important task, Experienceprofessionals can minimize data redundancy and perform fine tuning which reducesretrieval time.

5. **Concurrent access and Crash recovery:**

DBMS schedules concurrent access to the data. DBMS protects user from the effects ofsystem failure.

6. **Reduced application development time**.

DBMS supports important functions that are common to many applications



Architecture of DBMS

****

A commonly used views of data approach is the three-level architecture suggested byANSI/SPARC (American National Standards Institute/Standards Planning andRequirements Committee). ANSI/SPARC produced an interim report in 1972 followedby a final report in 1977. The reports proposed an architectural framework for databases.Under this approach, a database is considered as containing data about an *enterprise*. Thethree levels of the architecture are three different views of the data:

**Levels of abstraction in DBMS**

**External - individual user view**

**Conceptual - community user view**

**Internal - physical or storage view**

Schema: A description of data in terms of a data model is called a schema. In the relational model, the schema for a relation specifies its name, the name of each field, and the type of each field. As an example

Employee(eid: string, ename: string, street:string, city:real)

WORKS ( eid:string, <cid:string>, salary:real)

COMPANY(<cid:string>, company-name:string,city:string)

MANAGERS (mid:string, eid:string, <cid:string>, manager-name:string)

**Conceptual schema**:

Also called as logical schema describes the stored data in terms of the data model of the RDBMS. This describes all relations that are stored in the database.

**Physical Schema:**

This specifies the storage details. This shows how the relations described in conceptual schema are actually stored on secondary storage devices. It describes about what file organizations is used, which data structures are used, usage of indexes to retrieve the speedy retravel operations.

**External Schema:**

External schemas allow data access to be customized at the level of individual users or groups.Any given database has exactly one conceptual schema and one physical schema because it has just one set of stored relations, but it may have several external schemas, each tailored to a particular group of users. Each external schema consists of a collection of one or more views and relations from the conceptual schema. A view is conceptually a relation, but the records in a view are not stored in the DBMS.

The three level database architecture allows a clear separation of the information meaning(conceptual view) from the external data representation and from the physical datastructure layout. A database system that is able to separate the three different views ofdata is likely to be flexible and adaptable. This flexibility and adaptability is dataindependence that we have discussed earlier.

The external level is the view that the individual user of the database has. This view isoften a restricted view of the database and the same database may provide a number ofdifferent views for different classes of users. In general, the end users and even the application programmers are only interested in a subset of the database. For example, adepartment head may only be interested in the departmental finances and studentenrolments but not the library information. The librarian would not be expected to haveany interest in the information about academic staff. The payroll office would have nointerest in student enrolments.

The conceptual view is the information model of the enterprise and contains the view ofthe whole enterprise without any concern for the physical implementation. This view isnormally more stable than the other two views. In a database, it may be desirable tochange the internal view to improve performance while there has been no change in theconceptual view of the database. The conceptual view is the overall community view ofthe database and it includes all the information that is going to be represented in thedatabase. The conceptual view is defined by the conceptual schema which includesdefinitions of each of the various types of data.

The internal view is the view about the actual physical storage of data. It tells us whatdata is stored in the database and how. At least the following aspects are considered atthis level:

Storage allocation e.g. B-trees, hashing etc.

Access paths e.g. specification of primary and secondary keys, indexes and

pointers and sequencing.

Miscellaneous e.g. data compression and encryption techniques, optimization of

the internal structures.

Efficiency considerations are the most important at this level and the datastructures are chosen to provide an efficient database. The internal view does not dealwith the physical devices directly. Instead it views a physical device as a collection ofphysical pages and allocates space in terms of logical pages.

The separation of the conceptual view from the internal view enables us toprovide a logical description of the database without the need to specify physicalstructures. This is often called *physical data independence*. Separating the external viewsfrom the conceptual view enables us to change the conceptual view without affecting theexternal views. This separation is sometimes called *logical data independence*.

Assuming the three level view of the database, a number of mappings are needed toenable the users working with one of the external views. For example, the payroll officemay have an external view of the database that consists of the following informationonly:

Staff number, name and address.

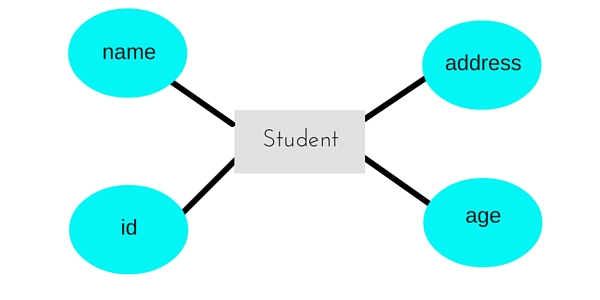
Staff tax information e.g. number of dependents.

Staff bank information where salary is deposited.

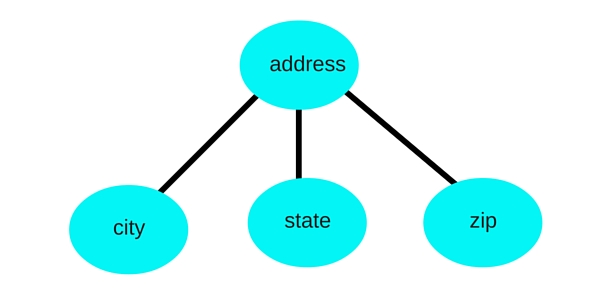
Staff employment status, salary level, leave information etc.

The conceptual view of the database may contain academic staff, general staff, casualstaff etc. A mapping will need to be created where all the staff in the different categoriesare combined into one category for the payroll office. The conceptual view would includeinformation about each staff's position, the date employment started, full-time or part-timeetc. This will need to be mapped to the salary level for the salary office. Also, ifthere is some change in the conceptual view, the external view can stay the same if themapping is changed.

Attribute example



Composite attribute



Metadata

**Metadata** characterizes data. It is used to provide documentation such that data can be understood and more readily consumed by your organization. Metadata answers the ***who***, ***what***, ***when***, ***where***, ***why***, and **how** questions for users of the data.In addition to managing data, corporations must to be able to manage and control the definition of the data elements used in databases. Without an understanding of the structure, limitations, definition, and description of data, it is likely that data will be misinterpreted or misused; further, data that is not well-defined can cause database integrity problems. This is a metadata issue.

In order for data to be anything more than simply data, metadata is required. Without metadata, data has no identifiable meaning – it is merely a collection of digits, characters, or bits. Metadata gives data its form and makes it usable by information professionals.

most DBMSs store all of the following metadata in the system catalog:

* The names of every database, table, column, index, view, relationship, stored procedure, trigger, and so on.
* The primary key for each table and any foreign keys that refer back to that primary key.
* Which tables are in which views.
* The data type, length, and constraints for each column of every table.
* The names of the physical files used to store database data, as well as information about file storage, extents, and disk volumes.
* Authorization and security information detailing which users have what type of authority on which database objects.
* The date and time of the last database definition change, as well as the ID of the user who implemented the DDL for the change.
* Database organization information.

For example, a [digital image](https://en.wikipedia.org/wiki/Digital_image) may include metadata that describes how large the picture is, the color depth, the image resolution, when the image was created, the shutter speed, and other data. A text document's metadata may contain information about how long the document is, who the author is, when the document was written, and a short summary of the document. Metadata within web pages can also contain descriptions of page content, as well as key words linked to the content. These links are often called "Metatags", which were used as the primary factor in determining order for a web search until the late 1990s. The reliance of metatags in web searches was decreased in the late 1990s because of "keyword stuffing". Metatags were being largely misused to trick search engines into thinking some websites had more relevance in the search than they really did.[Web pages](https://en.wikipedia.org/wiki/Web_page) often include metadata in the form of [meta tags](https://en.wikipedia.org/wiki/Meta_element). Description and keywords in meta tags are commonly used to describe the Web page's content. Meta elements also specify page description, key words, authors of the document, and when the document was last modified. Web page metadata helps search engines and users to find the types of web pages they are looking for.

# Lecture Notes: Relational Algebra

## Set operations

Relations in relational algebra are seen as sets of tuples, so we can use basic set operations.

## Projection

Example: The table **E** (for **EMPLOYEE**)

|  |  |  |  |
| --- | --- | --- | --- |
| **nr** | **name** | **salary** |  |
| 1 | John | 100 |  |
| 5 | Sarah | 300 |  |
| 7 | Tom | 100 |  |

|  |  |  |
| --- | --- | --- |
| **SQL** | **Result** | **Relational algebra** |
| select salary  from E | |  | | --- | | **salary** | | 100 | | 300 | | **PROJECT**salary(E) |
| select nr, salary  from E | |  |  | | --- | --- | | **nr** | **salary** | | 1 | 100 | | 5 | 300 | | 7 | 100 | | **PROJECT**nr, salary(E) |

Note that there are no duplicate rows in the result.

## Selection

The same table **E** (for **EMPLOYEE**) as above.

|  |  |  |
| --- | --- | --- |
| **SQL** | **Result** | **Relational algebra** |
| select \*  from E  where salary < 200 | |  |  |  | | --- | --- | --- | | **nr** | **name** | **salary** | | 1 | John | 100 | | 7 | Tom | 100 | | **SELECT**salary < 200(E) |
| select \*  from E  where salary < 200  and nr >= 7 | |  |  |  | | --- | --- | --- | | **nr** | **name** | **salary** | | 7 | Tom | 100 | | **SELECT**salary < 200 and nr >= 7(E) |

## Relational algebra expressions

|  |  |  |
| --- | --- | --- |
| **SQL** | **Result** | **Relational algebra** |
| select name, salary  from E  where salary < 200 | |  |  | | --- | --- | | **name** | **salary** | | John | 100 | | Tom | 100 | | **PROJECT**name, salary (**SELECT**salary < 200(E))  *or, step by step, using an intermediate result*  Temp <- **SELECT**salary < 200(E)  Result <- **PROJECT**name, salary(Temp) |

## Notation

The operations have their own symbols.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | | **Operation** | **My HTML** | **Symbol** | | Projection | **PROJECT** | [Greek letter pi](http://www.databasteknik.se/webbkursen/relalg-lecture/huge-pi.gif) | | Selection | **SELECT** | [Greek letter sigma](http://www.databasteknik.se/webbkursen/relalg-lecture/huge-sigma.gif) | | Renaming | **RENAME** | [Greek letter rho](http://www.databasteknik.se/webbkursen/relalg-lecture/huge-rho.gif) | | Union | **UNION** | [Union symbol](http://www.databasteknik.se/webbkursen/relalg-lecture/huge-union.gif) | | Intersection | **INTERSECTION** | [Intersection symbol](http://www.databasteknik.se/webbkursen/relalg-lecture/huge-intersection.gif) | | Assignment | **<-** | [Assignment symbol](http://www.databasteknik.se/webbkursen/relalg-lecture/huge-left-arrow.gif) | |  | |  |  |  | | --- | --- | --- | | **Operation** | **My HTML** | **Symbol** | | Cartesian product | **X** | [Symbol for Cartesian product](http://www.databasteknik.se/webbkursen/relalg-lecture/huge-cross.gif) | | Join | **JOIN** | [Join symbol](http://www.databasteknik.se/webbkursen/relalg-lecture/huge-join.gif) | | Left outer join | **LEFT OUTER JOIN** | [Symbol for left outer join](http://www.databasteknik.se/webbkursen/relalg-lecture/huge-left-outer-join.gif) | | Right outer join | **RIGHT OUTER JOIN** | [Symbol for right outer join](http://www.databasteknik.se/webbkursen/relalg-lecture/huge-right-outer-join.gif) | | Full outer join | **FULL OUTER JOIN** | [Symbol for full outer join](http://www.databasteknik.se/webbkursen/relalg-lecture/huge-full-outer-join.gif) | | Semijoin | **SEMIJOIN** | [Semijoin symbol](http://www.databasteknik.se/webbkursen/relalg-lecture/huge-semijoin.gif) | |

Example:

**PROJECTNamn( SELECTMedlemsnummer< 3 ( Medlem ) )**

should actually be written

[Ett relationsalgebrauttryck](http://www.databasteknik.se/webbkursen/relalg-lecture/large-expression.gif)

## Cartesian product

The *cartesian product* of two tables combines each row in one table with each row in the other table.

Example: The table **E** (for **EMPLOYEE**)

|  |  |  |
| --- | --- | --- |
| **enr** | **ename** | **dept** |
| 1 | Bill | A |
| 2 | Sarah | C |
| 3 | John | A |

Example: The table **D** (for **DEPARTMENT**)

|  |  |
| --- | --- |
| **dnr** | **dname** |
| A | Marketing |
| B | Sales |
| C | Legal |

|  |  |  |
| --- | --- | --- |
| **SQL** | **Result** | **Relational algebra** |
| select \*  from E, D | |  |  |  |  |  | | --- | --- | --- | --- | --- | | **enr** | **ename** | **dept** | **dnr** | **dname** | | 1 | Bill | A | A | Marketing | | 1 | Bill | A | B | Sales | | 1 | Bill | A | C | Legal | | 2 | Sarah | C | A | Marketing | | 2 | Sarah | C | B | Sales | | 2 | Sarah | C | C | Legal | | 3 | John | A | A | Marketing | | 3 | John | A | B | Sales | | 3 | John | A | C | Legal | | E **X** D |

* Seldom useful in practice.
* Usually an error.
* Can give a huge result.

## Join (sometimes called "inner join")

The cartesian product example above combined each employee with each department. If we only keep those lines where the **dept** attribute for the employee is equal to the **dnr** (the department number) of the department, we get a nice list of the employees, and the department that each employee works for:

|  |  |  |
| --- | --- | --- |
| **SQL** | **Result** | **Relational algebra** |
| select \*  from E, D  where dept = dnr | |  |  |  |  |  | | --- | --- | --- | --- | --- | | **enr** | **ename** | **dept** | **dnr** | **dname** | | 1 | Bill | A | A | Marketing | | 2 | Sarah | C | C | Legal | | 3 | John | A | A | Marketing | | **SELECT**dept = dnr (E **X** D)  *or, using the equivalent join operation*  E **JOIN**dept = dnr D |

* A very common and useful operation.
* Equivalent to a cartesian product followed by a select.
* Inside a relational DBMS, it is usually much more efficient to calculate a join directly, instead of calculating a cartesian product and then throwing away most of the lines.
* Note that the same SQL query can be translated to several different relational algebra expressions, which all give the same result.
* If we assume that these relational algebra expressions are executed, inside a relational DBMS which uses relational algebra operations as its lower-level internal operations, different relational algebra expressions can take very different time (and memory) to execute.

## Natural join

A normal inner join, but using the join condition that columns with the same names should be equal. Duplicate columns are removed.

## Renaming tables and columns

Example: The table **E** (for **EMPLOYEE**)

|  |  |  |
| --- | --- | --- |
| **nr** | **name** | **dept** |
| 1 | Bill | A |
| 2 | Sarah | C |
| 3 | John | A |

Example: The table **D** (for **DEPARTMENT**)

|  |  |
| --- | --- |
| **nr** | **name** |
| A | Marketing |
| B | Sales |
| C | Legal |

We want to join these tables, but:

* Several columns in the result will have the same name (**nr** and **name**).
* How do we express the join condition, when there are two columns called **nr**?

Solutions:

* Rename the attributes, using the *rename* operator.
* Keep the names, and prefix them with the table name, as is done in SQL. (This is somewhat unorthodox.)

|  |  |  |
| --- | --- | --- |
| **SQL** | **Result** | **Relational algebra** |
| select \*  from E as E(enr, ename, dept),  D as D(dnr, dname)  where dept = dnr | |  |  |  |  |  | | --- | --- | --- | --- | --- | | **enr** | **ename** | **dept** | **dnr** | **dname** | | 1 | Bill | A | A | Marketing | | 2 | Sarah | C | C | Legal | | 3 | John | A | A | Marketing | | (**RENAME**(enr, ename, dept)(E)) **JOIN**dept = dnr (**RENAME**(dnr, dname)(D)) |
| select \*  from E, D  where dept = D.nr | |  |  |  |  |  | | --- | --- | --- | --- | --- | | **nr** | **name** | **dept** | **nr** | **name** | | 1 | Bill | A | A | Marketing | | 2 | Sarah | C | C | Legal | | 3 | John | A | A | Marketing | | E **JOIN**dept = D.nr D |

You can use another variant of the renaming operator to change the name of a table, for example to change the name of **E** to **R**. This is necessary when joining a table with itself (see below).

**RENAME**R(E)

A third variant lets you rename both the table and the columns:

**RENAME**R(enr, ename, dept)(E)

## Aggregate functions

Example: The table **E** (for **EMPLOYEE**)

|  |  |  |  |
| --- | --- | --- | --- |
| **nr** | **name** | **salary** | **dept** |
| 1 | John | 100 | A |
| 5 | Sarah | 300 | C |
| 7 | Tom | 100 | A |
| 12 | Anne | **null** | C |

|  |  |  |
| --- | --- | --- |
| **SQL** | **Result** | **Relational algebra** |
| select sum(salary)  from E | |  | | --- | | **sum** | | 500 | | Fsum(salary)(E) |

Note:

* Duplicates are not eliminated.
* **Null** values are ignored.

|  |  |  |
| --- | --- | --- |
| **SQL** | **Result** | **Relational algebra** |
| select count(salary)  from E | Result:   |  | | --- | | **count** | | 3 | | Fcount(salary)(E) |
| select count(distinct salary)  from E | Result:   |  | | --- | | **count** | | 2 | | Fcount(salary)(**PROJECT**salary(E)) |

You can calculate aggregates "grouped by" something:

|  |  |  |
| --- | --- | --- |
| **SQL** | **Result** | **Relational algebra** |
| select sum(salary)  from E  group by dept | |  |  | | --- | --- | | **dept** | **sum** | | A | 200 | | C | 300 | | deptFsum(salary)(E) |

Several aggregates simultaneously:

|  |  |  |
| --- | --- | --- |
| **SQL** | **Result** | **Relational algebra** |
| select sum(salary), count(\*)  from E  group by dept | |  |  |  | | --- | --- | --- | | **dept** | **sum** | **count** | | A | 200 | 2 | | C | 300 | 1 | | deptFsum(salary), count(\*)(E) |

Standard aggregate functions: sum, count, avg, min, max

## Hierarchies

Example: The table **E** (for **EMPLOYEE**)

|  |  |  |
| --- | --- | --- |
| **nr** | **name** | **mgr** |
| 1 | Gretchen | **null** |
| 2 | Bob | 1 |
| 5 | Anne | 2 |
| 6 | John | 2 |
| 3 | Hulda | 1 |
| 4 | Hjalmar | 1 |
| 7 | Usama | 4 |

## Outer join

Example: The table **E** (for **EMPLOYEE**)

|  |  |  |
| --- | --- | --- |
| **enr** | **ename** | **dept** |
| 1 | Bill | A |
| 2 | Sarah | C |
| 3 | John | A |

Example: The table **D** (for **DEPARTMENT**)

|  |  |
| --- | --- |
| **dnr** | **dname** |
| A | Marketing |
| B | Sales |
| C | Legal |

List each employee together with the department he or she works at:

|  |  |  |
| --- | --- | --- |
| **SQL** | **Result** | **Relational algebra** |
| select \*  from E, D  where edept = dnr  *or, using an explicit join*  select \*  from (E join D on edept = dnr) | |  |  |  |  |  | | --- | --- | --- | --- | --- | | **enr** | **ename** | **dept** | **dnr** | **dname** | | 1 | Bill | A | A | Marketing | | 2 | Sarah | C | C | Legal | | 3 | John | A | A | Marketing | | E **JOIN**edept = dnr D |

No employee works at department B, Sales, so it is not present in the result. This is probably not a problem in this case. But what if we want to know the number of employees at each department?

|  |  |  |
| --- | --- | --- |
| **SQL** | **Result** | **Relational algebra** |
| select dnr, dname, count(\*)  from E, D  where edept = dnr  group by dnr, dname  *or, using an explicit join*  select dnr, dname, count(\*)  from (E join D on edept = dnr)  group by dnr, dname | |  |  |  | | --- | --- | --- | | **dnr** | **dname** | **count** | | A | Marketing | 2 | | C | Legal | 1 | | dnr, dname**F**count(\*)(E **JOIN**edept = dnr D) |

No employee works at department B, Sales, so it is not present in the result. It disappeared already in the join, so the aggregate function never sees it. But what if we want it in the result, with the right number of employees (zero)?

Use a *right outer join*, which keeps all the rows from the right table. If a row can't be connected to any of the rows from the left table according to the join condition, **null** values are used:

|  |  |  |
| --- | --- | --- |
| **SQL** | **Result** | **Relational algebra** |
| select \*  from (E right outer join D on edept = dnr) | |  |  |  |  |  | | --- | --- | --- | --- | --- | | **enr** | **ename** | **dept** | **dnr** | **dname** | | 1 | Bill | A | A | Marketing | | 2 | Sarah | C | C | Legal | | 3 | John | A | A | Marketing | | **null** | **null** | **null** | B | Sales | | E **RIGHT OUTER JOIN**edept = dnr D |
| select dnr, dname, count(\*)  from (E right outer join D on edept = dnr)  group by dnr, dname | |  |  |  | | --- | --- | --- | | **dnr** | **dname** | **count** | | A | Marketing | 2 | | B | Sales | 1 | | C | Legal | 1 | | dnr, dname**F**count(\*)(E **RIGHT OUTER JOIN**edept = dnr D) |
| select dnr, dname, count(enr)  from (E right outer join D on edept = dnr)  group by dnr, dname | |  |  |  | | --- | --- | --- | | **dnr** | **dname** | **count** | | A | Marketing | 2 | | B | Sales | 0 | | C | Legal | 1 | | dnr, dname**F**count(enr)(E **RIGHT OUTER JOIN**edept = dnr D) |

Join types:

* **JOIN** = "normal" join = inner join
* **LEFT OUTER JOIN** = left outer join
* **RIGHT OUTER JOIN** = right outer join
* **FULL OUTER JOIN** = full outer join

## Outer union

*Outer union* can be used to calculate the union of two relations that are *partially union compatible*. Not very common.

Example: The table **R**

|  |  |
| --- | --- |
| **A** | **B** |
| 1 | 2 |
| 3 | 4 |

Example: The table **S**

|  |  |
| --- | --- |
| **B** | **C** |
| 4 | 5 |
| 6 | 7 |

The result of an outer union between R and S:

|  |  |  |
| --- | --- | --- |
| **A** | **B** | **C** |
| 1 | 2 | **null** |
| 3 | 4 | **5** |
| **null** | 6 | **7** |

Top of Form

Bottom of Form

Top of Form

Bottom of Form

**Normalization of Database**

Database Normalisation is a technique of organizing the data in the database. Normalization is a systematic approach of decomposing tables to eliminate data redundancy and undesirable characteristics like Insertion, Update and Deletion Anomalies. It is a multi-step process that puts data into tabular form by removing duplicated data from the relation tables.

Normalization is used for mainly two purpose,

* Eliminating reduntant(useless) data.
* Ensuring data dependencies make sense i.e data is logically stored.

#### Problem Without Normalization

Without Normalization, it becomes difficult to handle and update the database, without facing data loss. Insertion, Updation and Deletion Anamolies are very frequent if Database is not Normalized. To understand these anomalies let us take an example of **Student** table.

|  |  |  |  |
| --- | --- | --- | --- |
| **S\_id** | **S\_Name** | **S\_Address** | **Subject\_opted** |
| 401 | Adam | Noida | Bio |
| 402 | Alex | Panipat | Maths |
| 403 | Stuart | Jammu | Maths |
| 404 | Adam | Noida | Physics |

* **Updation Anamoly :** To update address of a student who occurs twice or more than twice in a table, we will have to update **S\_Address** column in all the rows, else data will become inconsistent.
* **Insertion Anamoly :** Suppose for a new admission, we have a Student id(S\_id), name and address of a student but if student has not opted for any subjects yet then we have to insert **NULL** there, leading to Insertion Anamoly.
* **Deletion Anamoly :** If (S\_id) 401 has only one subject and temporarily he drops it, when we delete that row, entire student record will be deleted along with it.

#### Normalization Rule

Normalization rule are divided into following normal form.

1. First Normal Form
2. Second Normal Form
3. Third Normal Form
4. BCNF

#### First Normal Form (1NF)

As per First Normal Form, no two Rows of data must contain repeating group of information i.e each set of column must have a unique value, such that multiple columns cannot be used to fetch the same row. Each table should be organized into rows, and each row should have a primary key that distinguishes it as unique.

The **Primary key** is usually a single column, but sometimes more than one column can be combined to create a single primary key. For example consider a table which is not in First normal form

**Student Table :**

|  |  |  |
| --- | --- | --- |
| **Student** | **Age** | **Subject** |
| Adam | 15 | Biology, Maths |
| Alex | 14 | Maths |
| Stuart | 17 | Maths |

In First Normal Form, any row must not have a column in which more than one value is saved, like separated with commas. Rather than that, we must separate such data into multiple rows.

**Student Table following 1NF will be :**

|  |  |  |
| --- | --- | --- |
| **Student** | **Age** | **Subject** |
| Adam | 15 | Biology |
| Adam | 15 | Maths |
| Alex | 14 | Maths |
| Stuart | 17 | Maths |

Using the First Normal Form, data redundancy increases, as there will be many columns with same data in multiple rows but each row as a whole will be unique.

#### Second Normal Form (2NF)

As per the Second Normal Form there must not be any partial dependency of any column on primary key. It means that for a table that has concatenated primary key, each column in the table that is not part of the primary key must depend upon the entire concatenated key for its existence. If any column depends only on one part of the concatenated key, then the table fails **Second normal form**.

In example of First Normal Form there are two rows for Adam, to include multiple subjects that he has opted for. While this is searchable, and follows First normal form, it is an inefficient use of space. Also in the above Table in First Normal Form, while the candidate key is {**Student**, **Subject**}, **Age** of Student only depends on Student column, which is incorrect as per Second Normal Form. To achieve second normal form, it would be helpful to split out the subjects into an independent table, and match them up using the student names as foreign keys.

**New Student Table following 2NF will be :**

|  |  |
| --- | --- |
| **Student** | **Age** |
| Adam | 15 |
| Alex | 14 |
| Stuart | 17 |

In Student Table the candidate key will be **Student** column, because all other column i.e **Age** is dependent on it.

**New Subject Table introduced for 2NF will be :**

|  |  |
| --- | --- |
| **Student** | **Subject** |
| Adam | Biology |
| Adam | Maths |
| Alex | Maths |
| Stuart | Maths |

In Subject Table the candidate key will be {**Student**, **Subject**} column. Now, both the above tables qualifies for Second Normal Form and will never suffer from Update Anomalies. Although there are a few complex cases in which table in Second Normal Form suffers Update Anomalies, and to handle those scenarios Third Normal Form is there.

#### Third Normal Form (3NF)

**Third Normal form** applies that every non-prime attribute of table must be dependent on primary key, or we can say that, there should not be the case that a non-prime attribute is determined by another non-prime attribute. So this *transitive functional dependency* should be removed from the table and also the table must be in **Second Normal form**. For example, consider a table with following fields.

**Student\_Detail Table :**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Student\_id** | **Student\_name** | **DOB** | **Street** | **city** | **State** | **Zip** |

In this table Student\_id is Primary key, but street, city and state depends upon Zip. The dependency between zip and other fields is called **transitive dependency**. Hence to apply **3NF**, we need to move the street, city and state to new table, with **Zip** as primary key.

**New Student\_Detail Table :**

|  |  |  |  |
| --- | --- | --- | --- |
| **Student\_id** | **Student\_name** | **DOB** | **Zip** |

**Address Table :**

|  |  |  |  |
| --- | --- | --- | --- |
| **Zip** | **Street** | **city** | **state** |

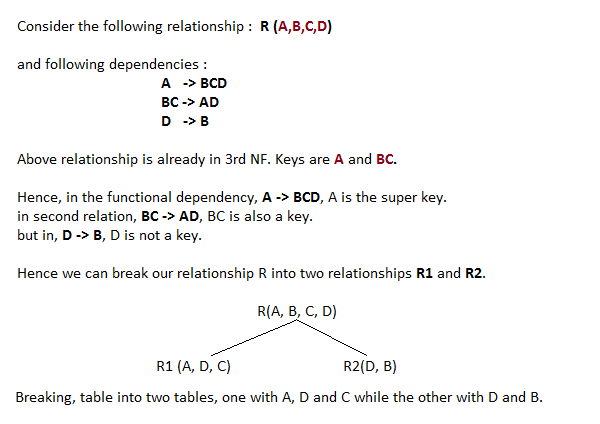
The advantage of removing transtive dependency is,

* Amount of data duplication is reduced.
* Data integrity achieved.

#### Boyce and Codd Normal Form (BCNF)

**Boyce and Codd Normal Form** is a higher version of the Third Normal form. This form deals with certain type of anamoly that is not handled by 3NF. A 3NF table which does not have multiple overlapping candidate keys is said to be in BCNF. For a table to be in BCNF, following conditions must be satisfied:

* R must be in 3rd Normal Form
* and, for each functional dependency ( X -> Y ), X should be a super Key.



**Example 2**

**Example**: Suppose a manufacturing company stores the employee details in a table named employee that has four attributes: emp\_id for storing employee’s id, emp\_name for storing employee’s name, emp\_address for storing employee’s address and emp\_dept for storing the department details in which the employee works. At some point of time the table looks like this:

|  |  |  |  |
| --- | --- | --- | --- |
| emp\_id | emp\_name | emp\_address | emp\_dept |
| 101 | Rick | Delhi | D001 |
| 101 | Rick | Delhi | D002 |
| 123 | Maggie | Agra | D890 |
| 166 | Glenn | Chennai | D900 |
| 166 | Glenn | Chennai | D004 |

The above table is not normalized. We will see the problems that we face when a table is not normalized.

**Update anomaly**: In the above table we have two rows for employee Rick as he belongs to two departments of the company. If we want to update the address of Rick then we have to update the same in two rows or the data will become inconsistent. If somehow, the correct address gets updated in one department but not in other then as per the database, Rick would be having two different addresses, which is not correct and would lead to inconsistent data.

**Insert anomaly**: Suppose a new employee joins the company, who is under training and currently not assigned to any department then we would not be able to insert the data into the table if emp\_dept field doesn’t allow nulls.

**Delete anomaly**: Suppose, if at a point of time the company closes the department D890 then deleting the rows that are having emp\_dept as D890 would also delete the information of employee Maggie since she is assigned only to this department.

To overcome these anomalies we need to normalize the data. In the next section we will discuss about normalization.

## Normalization

Here are the most commonly used normal forms:

* First normal form(1NF)
* Second normal form(2NF)
* Third normal form(3NF)
* Boyce & Codd normal form (BCNF)

## First normal form (1NF)

As per the rule of first normal form, an attribute (column) of a table cannot hold multiple values. It should hold only atomic values.

**Example**: Suppose a company wants to store the names and contact details of its employees. It creates a table that looks like this:

|  |  |  |  |
| --- | --- | --- | --- |
| emp\_id | emp\_name | emp\_address | emp\_mobile |
| 101 | Herschel | New Delhi | 8912312390 |
| 102 | Jon | Kanpur | 8812121212  9900012222 |
| 103 | Ron | Chennai | 7778881212 |
| 104 | Lester | Bangalore | 9990000123  8123450987 |

Two employees (Jon & Lester) are having two mobile numbers so the company stored them in the same field as you can see in the table above.

This table is **not in 1NF** as the rule says “each attribute of a table must have atomic (single) values”, the emp\_mobile values for employees Jon & Lester violates that rule.

To make the table complies with 1NF we should have the data like this:

|  |  |  |  |
| --- | --- | --- | --- |
| emp\_id | emp\_name | emp\_address | emp\_mobile |
| 101 | Herschel | New Delhi | 8912312390 |
| 102 | Jon | Kanpur | 8812121212 |
| 102 | Jon | Kanpur | 9900012222 |
| 103 | Ron | Chennai | 7778881212 |
| 104 | Lester | Bangalore | 9990000123 |
| 104 | Lester | Bangalore | 8123450987 |

## Second normal form (2NF)

A table is said to be in 2NF if both the following conditions hold:

* Table is in 1NF (First normal form)
* No non-prime attribute is dependent on the proper subset of any candidate key of table.

An attribute that is not part of any candidate key is known as non-prime attribute.

**Example**: Suppose a school wants to store the data of teachers and the subjects they teach. They create a table that looks like this: Since a teacher can teach more than one subjects, the table can have multiple rows for a same teacher.

|  |  |  |
| --- | --- | --- |
| teacher\_id | subject | teacher\_age |
| 111 | Maths | 38 |
| 111 | Physics | 38 |
| 222 | Biology | 38 |
| 333 | Physics | 40 |
| 333 | Chemistry | 40 |

**Candidate Keys**: {teacher\_id, subject}  
**Non prime attribute**: teacher\_age

The table is in 1 NF because each attribute has atomic values. However, it is not in 2NF because non prime attribute teacher\_age is dependent on teacher\_id alone which is a proper subset of candidate key. This violates the rule for 2NF as the rule says “**no** non-prime attribute is dependent on the proper subset of any candidate key of the table”.

To make the table complies with 2NF we can break it in two tables like this:  
**teacher\_details table:**

|  |  |
| --- | --- |
| teacher\_id | teacher\_age |
| 111 | 38 |
| 222 | 38 |
| 333 | 40 |

**teacher\_subject table:**

|  |  |
| --- | --- |
| teacher\_id | subject |
| 111 | Maths |
| 111 | Physics |
| 222 | Biology |
| 333 | Physics |
| 333 | Chemistry |

Now the tables comply with Second normal form (2NF).

## Third Normal form (3NF)

A table design is said to be in 3NF if both the following conditions hold:

* Table must be in 2NF
* [Transitive functional dependency](http://beginnersbook.com/2015/04/transitive-dependency-in-dbms/) of non-prime attribute on any super key should be removed.

An attribute that is not part of any [candidate key](http://beginnersbook.com/2015/04/candidate-key-in-dbms/) is known as non-prime attribute.

In other words 3NF can be explained like this: A table is in 3NF if it is in 2NF and for each functional dependency X-> Y at least one of the following conditions hold:

* X is a [super key](http://beginnersbook.com/2015/04/super-key-in-dbms/) of table
* Y is a prime attribute of table

An attribute that is a part of one of the candidate keys is known as prime attribute.

**Example**: Suppose a company wants to store the complete address of each employee, they create a table named employee\_details that looks like this:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| emp\_id | emp\_name | emp\_zip | emp\_state | emp\_city | emp\_district |
| 1001 | John | 282005 | UP | Agra | Dayal Bagh |
| 1002 | Ajeet | 222008 | TN | Chennai | M-City |
| 1006 | Lora | 282007 | TN | Chennai | Urrapakkam |
| 1101 | Lilly | 292008 | UK | Pauri | Bhagwan |
| 1201 | Steve | 222999 | MP | Gwalior | Ratan |

**Super keys**: {emp\_id}, {emp\_id, emp\_name}, {emp\_id, emp\_name, emp\_zip}…so on  
**Candidate Keys**: {emp\_id}  
**Non-prime attributes**: all attributes except emp\_id are non-prime as they are not part of any candidate keys.

Here, emp\_state, emp\_city & emp\_district dependent on emp\_zip. And, emp\_zip is dependent on emp\_id that makes non-prime attributes (emp\_state, emp\_city & emp\_district) transitively dependent on super key (emp\_id). This violates the rule of 3NF.

To make this table complies with 3NF we have to break the table into two tables to remove the transitive dependency:

**employee table:**

|  |  |  |
| --- | --- | --- |
| emp\_id | emp\_name | emp\_zip |
| 1001 | John | 282005 |
| 1002 | Ajeet | 222008 |
| 1006 | Lora | 282007 |
| 1101 | Lilly | 292008 |
| 1201 | Steve | 222999 |

**employee\_zip table:**

|  |  |  |  |
| --- | --- | --- | --- |
| emp\_zip | emp\_state | emp\_city | emp\_district |
| 282005 | UP | Agra | Dayal Bagh |
| 222008 | TN | Chennai | M-City |
| 282007 | TN | Chennai | Urrapakkam |
| 292008 | UK | Pauri | Bhagwan |
| 222999 | MP | Gwalior | Ratan |

## Boyce Codd normal form (BCNF)

It is an advance version of 3NF that’s why it is also referred as 3.5NF. BCNF is stricter than 3NF. A table complies with BCNF if it is in 3NF and for every [functional dependency](http://beginnersbook.com/2015/04/functional-dependency-in-dbms/) X->Y, X should be the super key of the table.

**Example**: Suppose there is a company wherein employees work in **more than one department**. They store the data like this:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| emp\_id | emp\_nationality | emp\_dept | dept\_type | dept\_no\_of\_emp |
| 1001 | Austrian | Production and planning | D001 | 200 |
| 1001 | Austrian | stores | D001 | 250 |
| 1002 | American | design and technical support | D134 | 100 |
| 1002 | American | Purchasing department | D134 | 600 |

**Functional dependencies in the table above**:  
emp\_id -> emp\_nationality  
emp\_dept -> {dept\_type, dept\_no\_of\_emp}

**Candidate key**: {emp\_id, emp\_dept}

The table is not in BCNF as neither emp\_id nor emp\_dept alone are keys.

To make the table comply with BCNF we can break the table in three tables like this:  
**emp\_nationality table:**

|  |  |
| --- | --- |
| emp\_id | emp\_nationality |
| 1001 | Austrian |
| 1002 | American |

**emp\_dept table:**

|  |  |  |
| --- | --- | --- |
| emp\_dept | dept\_type | dept\_no\_of\_emp |
| Production and planning | D001 | 200 |
| stores | D001 | 250 |
| design and technical support | D134 | 100 |
| Purchasing department | D134 | 600 |

**emp\_dept\_mapping table:**

|  |  |
| --- | --- |
| emp\_id | emp\_dept |
| 1001 | Production and planning |
| 1001 | stores |
| 1002 | design and technical support |
| 1002 | Purchasing department |

**Functional dependencies**:  
emp\_id -> emp\_nationality  
emp\_dept -> {dept\_type, dept\_no\_of\_emp}

**Candidate keys**:  
For first table: emp\_id  
For second table: emp\_dept  
For third table: {emp\_id, emp\_dept}

This is now in BCNF as in both the functional dependencies left side part is a key.

**Third example**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **rollno** | **name** | **branch** | **hod** | **office\_tel** |
| 401 | Akon | CSE | Mr. X | 53337 |
| 402 | Bkon | CSE | Mr. X | 53337 |
| 403 | Ckon | CSE | Mr. X | 53337 |
| 404 | Dkon | CSE | Mr. X | 53337 |

### In the above table all the students belong to CSE branch. In this the branch, hod, and office\_tel fields have the same data which is data redundancy.

### Insertion anamoly: when there is a new admission,the student data cannot be inserted unless the students opt for a branch is known.

### Updation anomaly

### What if Mr.X leaves teh college? Or no longer the HOD of computer science department? In which case all the students record has to be updated with new data. Even if one record is missed out from updation then this will lead to data inconsistency.

### Deletion anomaly

### If the students record is deleted, then we lose the branch details also.

## Normalization Rule

Normalization rules are divided into the following normal forms:

1. First Normal Form
2. Second Normal Form
3. Third Normal Form
4. BCNF
5. Fourth Normal Form

### First Normal Form (1NF)

For a table to be in the First Normal Form, it should follow the following 4 rules:

1. It should only have single(atomic) valued attributes/columns.
2. Values stored in a column should be of the same domain
3. All the columns in a table should have unique names.
4. And the order in which data is stored, does not matter.

### Rules for first normal form

### 1.single valued attributes

### Each column should be single valued i.e atomic values.

### 2. attribute domain should not change

### Ie. In the column designated for date of birth should not be used for storing name of a person. If a column is designated to store last name of a person, first name should not be stored.

### 3. Unique name for attribute/columns

### Each column should be unique.

### Order doesn’t matters

### This says that the data can be stored in any order.

|  |  |  |
| --- | --- | --- |
| **oll\_no** | **name** | **subject** |
| 101 | Akon | OS, CN |
| 103 | Ckon | Java |
| 102 | Bkon | C, C++ |

### In the above table rule 2,3,4 are satisfied but rule 1 is violated as subject column consists of multiple values.

### To satisfy this rule the multiple values are to be written as single values like below:

|  |  |  |
| --- | --- | --- |
| **roll\_no** | **name** | **subject** |
| 101 | Akon | OS |
| 101 | Akon | CN |
| 103 | Ckon | Java |
| 102 | Bkon | C |
| 102 | Bkon | C++ |

### 2nd normal form

For a table to be in the Second Normal Form, it must satisfy two conditions:

1. The table should be in the First Normal Form.
2. There should be no Partial Dependency.

### What is dependency?

Let's take an example of a **Student** table with columns **student\_id, name**, **reg\_no**(registration number), **branch** and **address**(student's home address).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **student\_id** | **name** | **reg\_no** | **branch** | **address** |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

In this table, **student\_id** is the primary key and will be unique for every row, hence we can use **student\_id** to fetch any row of data from this table

Even for a case, where student names are same, if we know the **student\_id** we can easily fetch the correct record.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **student\_id** | **name** | **reg\_no** | **branch** | **address** |
| 10 | Akon | 07-WY | CSE | Kerala |
| 11 | Akon | 08-WY | IT | Gujarat |

Hence we can say a **Primary Key** for a table is the column or a group of columns(composite key) which can uniquely identify each record in the table. If we have the student\_id then we will be able to extract that particular record. All the columns depent on student\_id (primary key). This is called **dependency** otherwise **functional dependency**.

What is Partial Dependency?

Consider another table which consists of subject details wher **subject\_id** is the primary key.

|  |  |
| --- | --- |
| **subject\_id** | **subject\_name** |
| 1 | Java |
| 2 | C++ |
| 3 | Php |

Let there be another table called Score, to store marks obtained by students in the respective subjects.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **score\_id** | **student\_id** | **subject\_id** | **marks** | **teacher** |
| 1 | 10 | 1 | 70 | Java Teacher |
| 2 | 10 | 2 | 75 | C++ Teacher |
| 3 | 11 | 1 | 80 | Java Teacher |

In the above table student\_id+subject\_id together form the primary key

In the above the non-key element techer depend only on subject\_id not on student\_id. This is Partial Dependency, where an attribute in a table depends on only a part of the primary key and not on the whole key.

How to remove partial dependency?

This can be done by removing teacher column from score table and keep it in subjet table as follows:

Subject table:

|  |  |  |
| --- | --- | --- |
| **subject\_id** | **subject\_name** | **teacher** |
| **1** | **Java** | **Java Teacher** |
| **2** | **C++** | **C++ Teacher** |
| **3** | **Php** | **Php Teacher** |

**Score table:**

|  |  |  |  |
| --- | --- | --- | --- |
| **score\_id** | **student\_id** | **subject\_id** | **marks** |
| 1 | 10 | 1 | 70 |
| 2 | 10 | 2 | 75 |
| 3 | 11 | 1 | 80 |

1. For a table to be in the Second Normal form, it should be in the First Normal form and it should not have Partial Dependency.
2. Partial Dependency exists, when for a composite primary key, any attribute in the table depends only on a part of the primary key and not on the complete primary key.
3. To remove Partial dependency, we can divide the table, remove the attribute which is causing partial dependency, and move it to some other table where it fits in well.

3rd Normal Form

In the **score**table , add two more column, **exam \_name** and **total\_marks**.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **score\_id** | **student\_id** | **subject\_id** | **marks** | **exam\_name** | **total\_marks** |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

Requirements for third normal form

1. It should be in the Second Normal form.
2. And it should not have Transitive Dependency.

What is transitive dependency?

When a non-prime attribute depends on other non-prime attributes rather than depending upon the prime attributes or primary key.

For example in the above table, the column **total\_marks** depends on **exam\_name** as with exam type the total score changes. For example, practicals are of less marks while theory exams are of more marks.

But, **exam\_name** is just another column in the score table. It is not a primary key or even a part of the primary key, and **total\_marks** depends on it. This is called transitive dependency.

This should be removed. How to remoce this? The columns exam\_name and total\_marks can be removed from Score table and put then in an Exam table and use exam\_id wherever required.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **score\_id** | **student\_id** | **subject\_id** | **marks** | **exam\_id** |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

**Exam** table

|  |  |  |
| --- | --- | --- |
| **exam\_id** | **exam\_name** | **total\_marks** |
| 1 | Workshop | 200 |
| 2 | Mains | 70 |
| 3 | Practicals | 30 |

**Advantage of removing Transitive Dependency**

The advantage of removing transitive dependency is,

* Amount of data duplication is reduced.
* Data integrity achieved.

## Rules for BCNF

For a table to satisfy the Boyce-Codd Normal Form, it should satisfy the following two conditions:

1. It should be in the **Third Normal Form**.
2. And, for any dependency A → B, A should be a **super key**.

The second point sounds a bit tricky, right? In simple words, it means, that for a dependency A → B, A cannot be a **non-prime attribute**, if B is a **prime attribute**.

Example

Enrollement table

|  |  |  |
| --- | --- | --- |
| **student\_id** | **subject** | **professor** |
| 101 | Java | P.Java |
| 101 | C++ | P.Cpp |
| 102 | Java | P.Java2 |
| 103 | C# | P.Chash |
| 104 | Java | P.Java |

In teh table above:

One student can select more than one subject

One professor can teach more than on subject

In teh above student\_id+subject together form the primary key.

One more important point to note here is, one professor teaches only one subject, but one subject may have two different professors.

Hence, there is a dependency between subject and professor here, where subject depends on the professor name.

This table satisfies the **1st Normal form** because all the values are atomic, column names are unique and all the values stored in a particular column are of same domain.

This table also satisfies the **2nd Normal Form** as their is no **Partial Dependency**.

And, there is no **Transitive Dependency**, hence the table also satisfies the **3rd Normal Form**.

But this table is not in **Boyce-Codd Normal Form**.

**Why this table is not in BCNF?**

In the table above, student\_id, subject form primary key, which means subjectcolumn is a **prime attribute**.

But, there is one more dependency, professor → subject.

And while subject is a prime attribute, professor is a **non-prime attribute**, which is not allowed by BCNF.

**How to satisfy BCNF?**

To make this relation(table) satisfy BCNF, we will decompose this table into two tables, **student** table and **professor** table.

Below we have the structure for both the tables.

**Student Table**

|  |  |
| --- | --- |
| **student\_id** | **p\_id** |
| 101 | 1 |
| 101 | 2 |
| and so on... | |

And, **Professor Table**

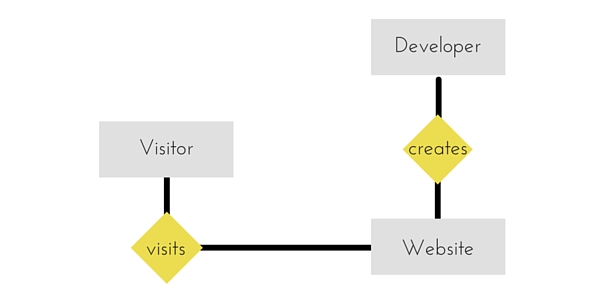
|  |  |  |
| --- | --- | --- |
| **p\_id** | **professor** | **subject** |
| 1 | P.Java | Java |
| 2 | P.Cpp | C++ |
| and so on... | | |

And now, this relation satisfy Boyce-Codd Normal Form. In the next tutorial we will learn about the **Fourth Normal Form**.

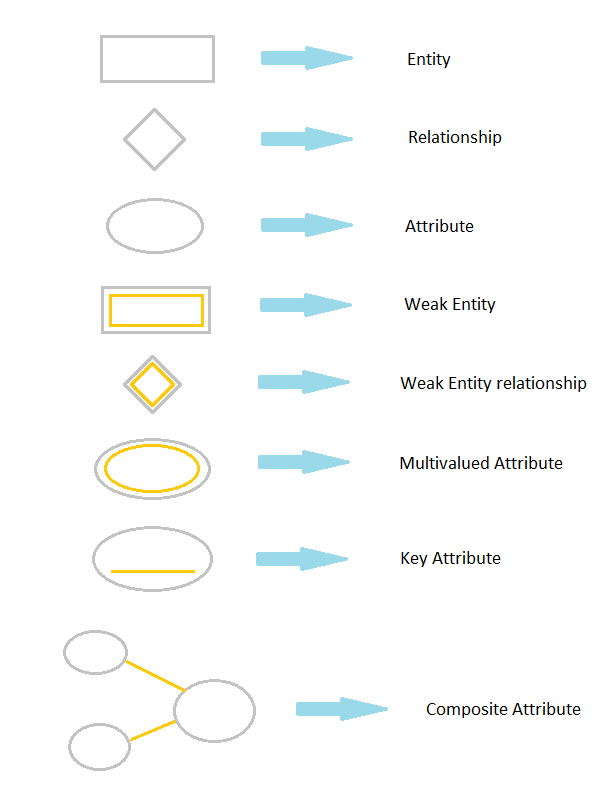
* Database designing is critical to the successful implementation of a database management system that meets the data requirements of an enterprise system.
* Normalization helps produce database systems that are cost-effective and have better security models.
* Functional dependencies are a very important component of the normalize data process
* Most database systems are normalized database up to the third normal forms.
* A primary uniquely identifies are record in a Table and cannot be null
* A foreign key helps connect table and references a primary key

### E-R Diagram

ER-Diagram is a visual representation of data that describes how data is related to each other.



#### Symbols and Notations

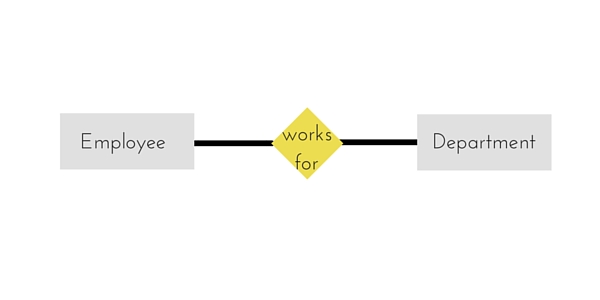


### Components of E-R Diagram

The E-R diagram has three main components.

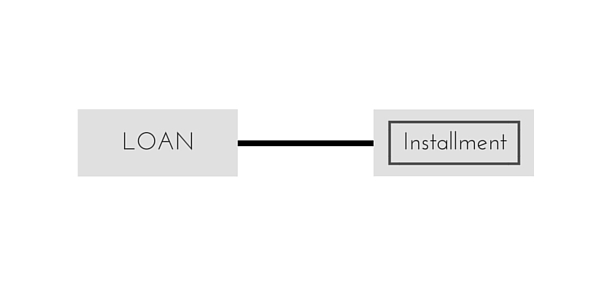
#### 1) Entity

An **Entity** can be any object, place, person or class. In E-R Diagram, an **entity** is represented using rectangles. Consider an example of an Organisation. Employee, Manager, Department, Product and many more can be taken as entities from an Organisation.



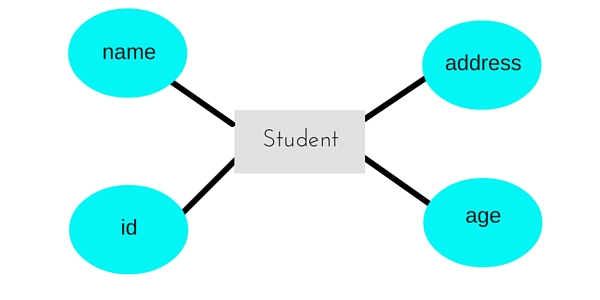
#### Weak Entity

Weak entity is an entity that depends on another entity. Weak entity doesn't have key attribute of their own. Double rectangle represents weak entity.



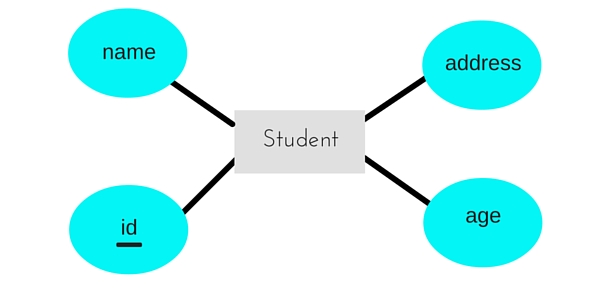
#### 2) Attribute

An **Attribute** describes a property or characteristic of an entity. For example, Name, Age, Address etc can be attributes of a Student. An attribute is represented using eclipse.



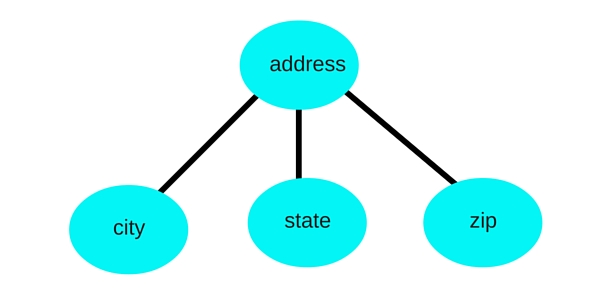
#### Key Attribute

Key attribute represents the main characteristic of an Entity. It is used to represent Primary key. Ellipse with underlying lines represent Key Attribute.



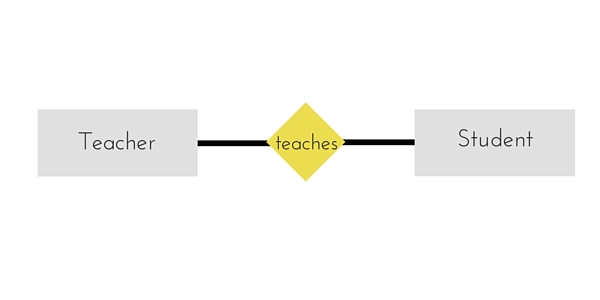
#### Composite Attribute

An attribute can also have their own attributes. These attributes are known as **Composite** attribute.



#### 3) Relationship

A Relationship describes relations between **entities**. Relationship is represented using diamonds.



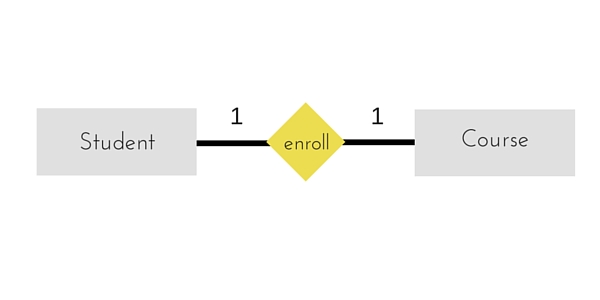
There are three types of relationship that exist between Entities.

* Binary Relationship
* Recursive Relationship
* Ternary Relationship

#### Binary Relationship

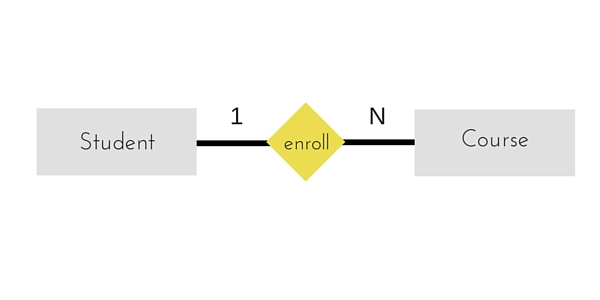
Binary Relationship means relation between two Entities. This is further divided into three types.

1. **One to One :** This type of relationship is rarely seen in real world.



The above example describes that one student can enroll only for one course and a course will also have only one Student. This is not what you will usually see in relationship.

1. **One to Many :** It reflects business rule that one entity is associated with many number of same entity. The example for this relation might sound a little weird, but this means that one student can enroll to many courses, but one course will have one Student.

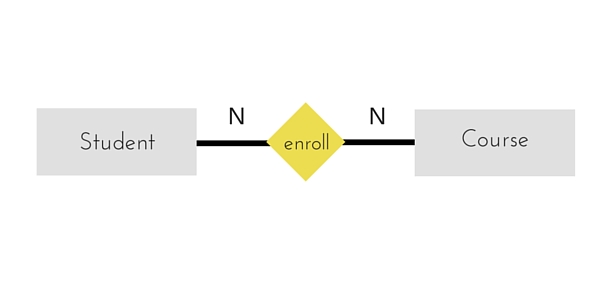


The arrows in the diagram describes that one student can enroll for only one course.

1. **Many to One :** It reflects business rule that many entities can be associated with just one entity. For example, Student enrolls for only one Course but a Course can have many Students.



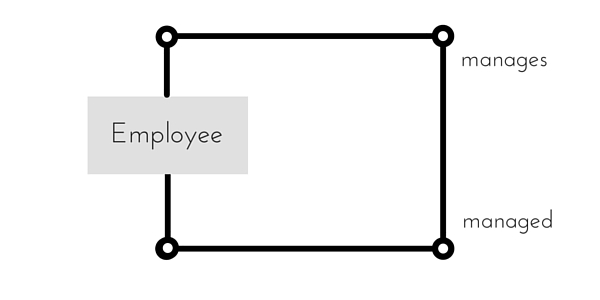
1. **Many to Many :**



The above diagram represents that many students can enroll for more than one courses.

#### Recursive Relationship

When an Entity is related with itself it is known as **Recursive** Relationship.

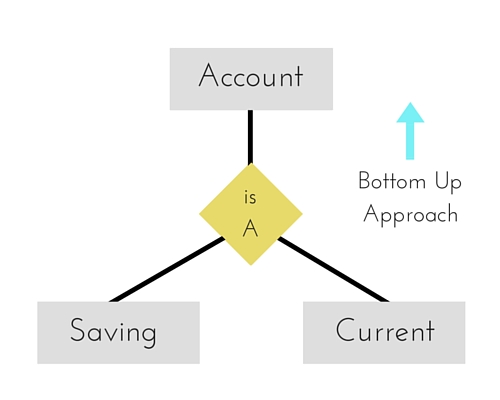


#### Ternary Relationship

Relationship of degree three is called Ternary relationship.

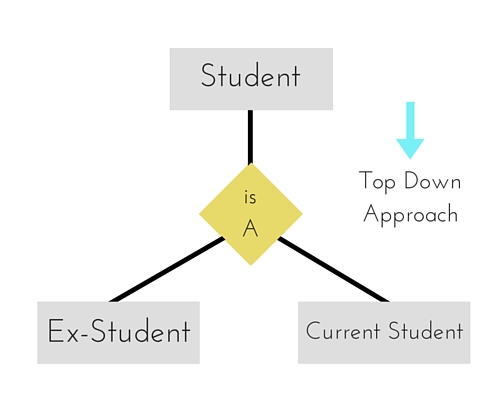
#### Generalization

**Generalization** is a bottom-up approach in which two lower level entities combine to form a higher level entity. In generalization, the higher level entity can also combine with other lower level entity to make further higher level entity.



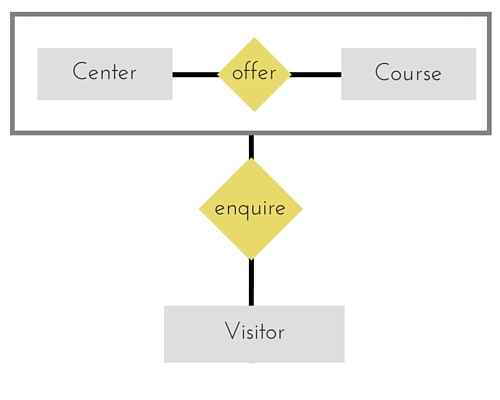
#### Specialization

**Specialization** is opposite to Generalization. It is a top-down approach in which one higher level entity can be broken down into two lower level entity. In specialization, some higher level entities may not have lower-level entity sets at all.



#### Aggregation

Aggregation is a process when relation between two entity is treated as a single entity. Here the relation between Center and Course, is acting as an Entity in relation with Visitor.



### Introduction to SQL

Structure Query Language(SQL) is a programming language used for storing and managing data in RDBMS. SQL was the first commercial language introduced for E.F Codd's **Relational** model. Today almost all RDBMS(MySql, Oracle, Infomix, Sybase, MS Access) uses **SQL** as the standard database language. SQL is used to perform all type of data operations in RDBMS.

#### SQL Command

SQL defines following data languages to manipulate data of RDBMS.

#### DDL : Data Definition Language

All DDL commands are auto-committed. That means it saves all the changes permanently in the database.

|  |  |
| --- | --- |
| **Command** | **Description** |
| create | to create new table or database |
| alter | for alteration |
| truncate | delete data from table |
| drop | to drop a table |
| rename | to rename a table |

#### DML : Data Manipulation Language

DML commands are not auto-committed. It means changes are not permanent to database, they can be rolled back.

|  |  |
| --- | --- |
| **Command** | **Description** |
| insert | to insert a new row |
| update | to update existing row |
| delete | to delete a row |
| merge | merging two rows or two tables |

#### TCL : Transaction Control Language

These commands are to keep a check on other commands and their affect on the database. These commands can annul changes made by other commands by rolling back to original state. It can also make changes permanent.

|  |  |
| --- | --- |
| **Command** | **Description** |
| commit | to permanently save |
| rollback | to undo change |
| savepoint | to save temporarily |

#### DCL : Data Control Language

Data control language provides command to grant and take back authority.

|  |  |
| --- | --- |
| **Command** | **Description** |
| grant | grant permission of right |
| revoke | take back permission. |

#### DQL : Data Query Language

|  |  |
| --- | --- |
| **Command** | **Description** |
| select | retrieve records from one or more table |

Naming rules

A table and column name can be up to 30 characters long and must start with alphabet only.

Letters (A-Z,a-z), numbers(0-9) and special characters $,\_,#, are allowed. Spaces and hypens are not allowed.

Data types

When a table is created, the columns must be specified with respective datatypes and the size of each column.

**Varchar2**

This type is a character data type to store variable-length alphanumeric data in a column. The default size is one character. The maximum allowable size is 8000 character. The size is specified within parenthesis-for example varchar2(20).

**Char**

The CHAR type is a character data type to store fixed-length alphanumeric data in a column. Minimum and default is one and maximum is 8000 characters. This is most appropriate when the columns are to be declared as fixed-length characters.

**Number**

This is used to store negative,positive, integer, fixed-decimal and floating point numbers. The size has to defined within parenthesis.

Example - Age number(3) – age can store 3 digit numerical integer value

Salary number(7,2) – salary stores a floating point number which has 7 digits of which two decimal places and one decimal point. Example 3456.78 ( in this total digits are 7 , 2 decimal digits, one decimal point, and 4 integer portion)

Date

The date data type is used for storing date and time values. The range of allowable dates is between January 1,4712 B.C and December 31, 9999 A.D. There is no need to specify the size for the Date column. Default format is DD-MON-YY.

The following table lists the general data types in SQL:

|  |  |
| --- | --- |
| **Data type** | **Description** |
| CHARACTER(n) | Character string. Fixed-length n |
| VARCHAR(n) or CHARACTER VARYING(n) | Character string. Variable length. Maximum length n |
| BINARY(n) | Binary string. Fixed-length n |
| BOOLEAN | Stores TRUE or FALSE values |
| VARBINARY(n) or BINARY VARYING(n) | Binary string. Variable length. Maximum length n |
| INTEGER(p) | Integer numerical (no decimal). Precision p |
| SMALLINT | Integer numerical (no decimal). Precision 5 |
| INTEGER | Integer numerical (no decimal). Precision 10 |
| BIGINT | Integer numerical (no decimal). Precision 19 |
| DECIMAL(p,s) | Exact numerical, precision p, scale s. Example: decimal(5,2) is a number that has 3 digits before the decimal and 2 digits after the decimal |
| NUMERIC(p,s) | Exact numerical, precision p, scale s. (Same as DECIMAL) |
| FLOAT(p) | Approximate numerical, mantissa precision p. A floating number in base 10 exponential notation. The size argument for this type consists of a single number specifying the minimum precision |
| REAL | Approximate numerical, mantissa precision 7 |
| FLOAT | Approximate numerical, mantissa precision 16 |
| DOUBLE PRECISION | Approximate numerical, mantissa precision 16 |
| DATE | Stores year, month, and day values |
| TIME | Stores hour, minute, and second values |
| TIMESTAMP | Stores year, month, day, hour, minute, and second values |
| INTERVAL | Composed of a number of integer fields, representing a period of time, depending on the type of interval |
| ARRAY | A set-length and ordered collection of elements |
| MULTISET | A variable-length and unordered collection of elements |
| XML | Stores XML data |

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. | |

**Note** − Here, datetime has 3.33 milliseconds accuracy where as smalldatetime has 1 minute accuracy.

Character Strings Data Types

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

Unicode Character Strings Data Types

|  |  |
| --- | --- |
| **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 Types

|  |  |
| --- | --- |
| **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) |

Misc 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 |

|  |  |
| --- | --- |
| **DDL -Data Definition Language:Command** | **Description** |
| CREATE | Creates a new table, a view of a table, or other object in database |
| ALTER | Modifies an existing database object, such as a table. |
| DROP | Deletes an entire table, a view of a table or other object in the database. |
| **DML -Data Manipulation Language:Command** | **Description** |
| INSERT | Creates a record |
| UPDATE | Modifies records |
| DELETE | Deletes records |
| **DCL -Data Control Language:Command** | **Description** |
| GRANT | Gives a privilege to user |
| REVOKE | Takes back privileges granted from user |
| **DQL -Data Query Language:Command** | **Description** |
| SELECT | Retrieves certain records from one or more tables |

What is an Operator in SQL?

An operator is a reserved word or a character used primarily in an SQL statement's WHERE clause to perform operation(s), such as comparisons and arithmetic operations.

Operators are used to specify conditions in an SQL statement and to serve as conjunctions for multiple conditions in a statement.

* Arithmetic operators
* Comparison operators
* Logical operators
* Operators used to negate conditions

SQL Arithmetic Operators:

Assume variable a holds 10 and variable b holds 20, then:

[Show Examples](http://www.tutorialspoint.com/sql/sql-arithmetic-operators.htm)

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| + | Addition - Adds values on either side of the operator | a + b will give 30 |
| - | Subtraction - Subtracts right hand operand from left hand operand | a - b will give -10 |
| \* | Multiplication - Multiplies values on either side of the operator | a \* b will give 200 |
| / | Division - Divides left hand operand by right hand operand | b / a will give 2 |
| % | Modulus - Divides left hand operand by right hand operand and returns remainder | b % a will give 0 |

SQL Comparison Operators:

Assume variable a holds 10 and variable b holds 20, then:

[Show Examples](http://www.tutorialspoint.com/sql/sql-comparison-operators.htm)

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| = | Checks if the values of two operands are equal or not, if yes then condition becomes true. | (a = b) is not true. |
| != | Checks if the values of two operands are equal or not, if values are not equal then condition becomes true. | (a != b) is true. |
| <> | Checks if the values of two operands are equal or not, if values are not equal then condition becomes true. | (a <> b) is true. |
| > | Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true. | (a > b) is not true. |
| < | Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true. | (a < b) is true. |
| >= | Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true. | (a >= b) is not true. |
| <= | Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true. | (a <= b) is true. |
| !< | Checks if the value of left operand is not less than the value of right operand, if yes then condition becomes true. | (a !< b) is false. |
| !> | Checks if the value of left operand is not greater than the value of right operand, if yes then condition becomes true. | (a !> b) is true. |

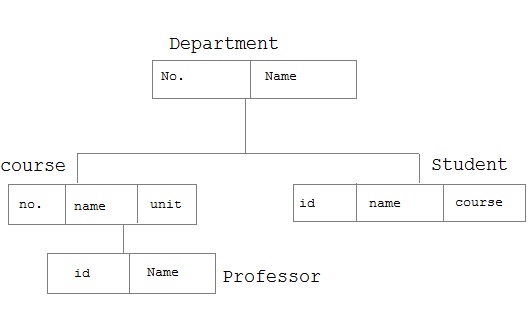
SQL Logical Operators:

Here is a list of all the logical operators available in SQL.

[Show Examples](http://www.tutorialspoint.com/sql/sql-logical-operators.htm)

|  |  |
| --- | --- |
| **Operator** | **Description** |
| ALL | The ALL operator is used to compare a value to all values in another value set. |
| AND | The AND operator allows the existence of multiple conditions in an SQL statement's WHERE clause. |
| ANY | The ANY operator is used to compare a value to any applicable value in the list according to the condition. |
| BETWEEN | The BETWEEN operator is used to search for values that are within a set of values, given the minimum value and the maximum value. |
| EXISTS | The EXISTS operator is used to search for the presence of a row in a specified table that meets certain criteria. |
| IN | The IN operator is used to compare a value to a list of literal values that have been specified. |
| LIKE | The LIKE operator is used to compare a value to similar values using wildcard operators. |
| NOT | The NOT operator reverses the meaning of the logical operator with which it is used. Eg: NOT EXISTS, NOT BETWEEN, NOT IN, etc. **This is a negate operator.** |
| OR | The OR operator is used to combine multiple conditions in an SQL statement's WHERE clause. |
| IS NULL | The NULL operator is used to compare a value with a NULL value. |
| UNIQUE | The UNIQUE operator searches every row of a specified table for uniqueness (no duplicates). |

Hierarchical Data model



**Types of constraints**

There are two types of constraints:

1. Integrity constraints: define both the primary key and the foreign key with the table and primary key it references. Integrity constraints are used to ensure accuracy and consistency of data in a relational database.
2. Value constraints: define if NULL values are disallowed, if UNIQUE values are required, and if only certain set of values are allowed in a column.

Naming a constraint:

The general convention used for naming constraints is

<table name><column name><constraint name>

It advisable to create constraints with name.Because if one wants to remove the constraints it can be done only if the constraints are declared with a name. A user cannot create constraints in two different tables with the same name.

|  |  |
| --- | --- |
| Constraint | Abbreviation |
| PRIMARY KEY | pk |
| FOREIGN KEY | fk |
| UNIQUE | uk |
| CHECK | ck or cc |
| NOT NULL | nn |

For example, a constraint name emp\_deptno\_fk means it is a constraint in table EMP on column Deptno of type foreign key.

Constraint can be created in two levels

1. Column level: A column level constraint references a single column and is defined along with the definition of column.
2. Table level : A table level constraint references one or more columns and is defined separately from the definitions of the columns.

**Primary key constraint**: This is known as entity integrity constraint.

The PRIMARY KEY constraint uniquely identifies each record in a database table.Primary keys must contain UNIQUE values.A primary key column cannot contain NULL values.

Most tables should have a primary key, and each table can have only ONE primary key.

**At the column level**:

DeptId NUMBER(2) CONSTRAINT dept\_deptid\_pk PRIMARY KEY

**At the table level**: if the table uses more than one key as its primary key ( i.e composite key) it can be declared only at the table level.

CONSTRAINT dependent\_emp\_dep\_pk PRIMARY KEY(employeeid, dependentid)

## SQL PRIMARY KEY Constraint on CREATE TABLE

The following SQL creates a PRIMARY KEY on the "rollno" column when the "student" table is created:

CREATE TABLE student  
(  
rollno int NOT NULL PRIMARY KEY,

LastName varchar(255) NOT NULL,  
FirstName varchar(255),  
Address varchar(255),  
City varchar(255)  
)

To allow naming of a PRIMARY KEY constraint, and for defining a PRIMARY KEY constraint on multiple columns, use the following SQL syntax:

CREATE TABLE student  
(  
rollno int NOT NULL,  
LastName varchar(255) NOT NULL,  
FirstName varchar(255),  
Address varchar(255),  
City varchar(255),  
CONSTRAINT student\_rollno\_pk PRIMARY KEY (rollno,LastName)  
)

**Note:** In the example above there is only ONE PRIMARY KEY (student\_rollno\_pk). However, the VALUE of the primary key is made up of TWO COLUMNS (rollno + LastName).

**SQL PRIMARY KEY Constraint on ALTER TABLE**

To create a PRIMARY KEY constraint on the "rollno" column when the table is already created, use the following SQL:

ALTER TABLE student  
ADD PRIMARY KEY (rollno)

To allow naming of a PRIMARY KEY constraint, and for defining a PRIMARY KEY constraint on multiple columns, use the following SQL syntax:

ALTER TABLE student

ADD CONSTRAINT student\_rollno\_pk PRIMARY KEY (rollno,LastName)

**Note:** If you use the ALTER TABLE statement to add a primary key, the primary key column(s) must already have been declared to not contain NULL values (when the table was first created).

**To DROP a PRIMARY KEY Constraint**

To drop a PRIMARY KEY constraint, use the following SQL:

ALTER TABLE student  
DROP CONSTRAINT student\_rollno\_pk

**SQL FOREIGN KEY Constraint**

A FOREIGN KEY in one table points to a PRIMARY KEY in another table.

Let's illustrate the foreign key with an example. Look at the following two tables:

The "Persons" table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **P\_Id** | **LastName** | **FirstName** | **Address** | **City** |
| 1 | Hansen | Ola | Timoteivn 10 | Sandnes |
| 2 | Svendson | Tove | Borgvn 23 | Sandnes |
| 3 | Pettersen | Kari | Storgt 20 | Stavanger |

The "Orders" table:

|  |  |  |
| --- | --- | --- |
| **O\_Id** | **OrderNo** | **P\_Id** |
| 1 | 77895 | 3 |
| 2 | 44678 | 3 |
| 3 | 22456 | 2 |
| 4 | 24562 | 1 |

Note that the "P\_Id" column in the "Orders" table points to the "P\_Id" column in the "Persons" table.

The "P\_Id" column in the "Persons" table is the PRIMARY KEY in the "Persons" table.

The "P\_Id" column in the "Orders" table is a FOREIGN KEY in the "Orders" table.

The FOREIGN KEY constraint is used to prevent actions that would destroy links between tables.

The FOREIGN KEY constraint also prevents invalid data from being inserted into the foreign key column, because it has to be one of the values contained in the table it points to.

**SQL FOREIGN KEY Constraint on CREATE TABLE**

The following SQL creates a FOREIGN KEY on the "P\_Id" column when the "Orders" table is created:

CREATE TABLE Orders  
(  
O\_Id int NOT NULL PRIMARY KEY,  
OrderNo int NOT NULL,  
P\_Id int FOREIGN KEY REFERENCES Persons(P\_Id)  
)

To allow naming of a FOREIGN KEY constraint, and for defining a FOREIGN KEY constraint on multiple columns, use the following SQL 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)  
)

SQL FOREIGN KEY Constraint on ALTER TABLE

To create a FOREIGN KEY constraint on the "P\_Id" column when the "Orders" table is already created, use the following SQL:

ALTER TABLE Orders  
ADD FOREIGN KEY (P\_Id)  
REFERENCES Persons(P\_Id)

To allow naming of a FOREIGN KEY constraint, and for defining a FOREIGN KEY constraint on multiple columns, use the following SQL syntax:

ALTER TABLE Orders  
ADD CONSTRAINT fk\_PerOrders  
FOREIGN KEY (P\_Id)  
REFERENCES Persons(P\_Id)

**To DROP a FOREIGN KEY Constraint**

To drop a FOREIGN KEY constraint, use the following SQL:

ALTER TABLE Orders  
DROP CONSTRAINT fk\_PerOrders

**SQL CHECK Constraint**

The CHECK constraint is used to limit the value range that can be placed in a column.

If you define a CHECK constraint on a single column it allows only certain values for this column.

If you define a CHECK constraint on a table it can limit the values in certain columns based on values in other columns in the row.

**SQL CHECK Constraint on CREATE TABLE**

The following SQL creates a CHECK constraint on the "P\_Id" column when the "Persons" table is created. The CHECK constraint specifies that the column "P\_Id" must only include integers greater than 0.

CREATE TABLE Persons  
(  
P\_Id int NOT NULL CHECK (P\_Id>0),  
LastName varchar(255) NOT NULL,  
FirstName varchar(255),  
Address varchar(255),  
City varchar(255)  
)

To allow naming of a CHECK constraint, and for defining a CHECK constraint on multiple columns, use the following SQL syntax:

CREATE TABLE Persons  
(  
P\_Id int NOT NULL,  
LastName varchar(255) NOT NULL,  
FirstName varchar(255),  
Address varchar(255),  
City varchar(255),  
CONSTRAINT chk\_Person CHECK (P\_Id>0 AND City='Sandnes')  
)

**SQL CHECK Constraint on ALTER TABLE**

To create a CHECK constraint on the "P\_Id" column when the table is already created, use the following SQL:

ALTER TABLE Persons  
ADD CHECK (P\_Id>0)

To allow naming of a CHECK constraint, and for defining a CHECK constraint on multiple columns, use the following SQL syntax:

ALTER TABLE Persons  
ADD CONSTRAINT chk\_Person CHECK (P\_Id>0 AND City='Sandnes')

Example : consider the following table

SQL> desc student1;

Name Null? Type

----------------------------------------- -------- -----------------------

SID NOT NULL CHAR(20)

NAME VARCHAR2(20)

LOGIN CHAR(20)

AGE NUMBER(38)

GPA FLOAT(63)

To add a check constraint:

SQL> alter table student1 add constraint mycheck check(age>=11);

Table altered.

When we try to enter for age below 11 the following error comes as follows:

SQL> insert into student1 values('107','kannan','kannan@cs',10,4.5);

insert into student1 values('107','kannan','kannan@cs',10,4.5)

\*

ERROR at line 1:

ORA-02290: check constraint (SYSTEM.MYCHECK) violated

**Now correctly entering data as follows:**

SQL> insert into student1 values('107','kannan','kannan@cs',16,4.5);

1 row created**.**

**To DROP a CHECK Constraint**

To drop a CHECK constraint, use the following SQL:

ALTER TABLE Persons  
DROP CONSTRAINT chk\_Person

## SQL NOT NULL Constraint

The NOT NULL constraint enforces a column to NOT accept NULL values.

The NOT NULL constraint enforces a field to always contain a value. This means that you cannot insert a new record, or update a record without adding a value to this field.

The following SQL enforces the "rollno" column and the "Name" column to not accept NULL values:

### Example

CREATE TABLE student  
(  
rollno int NOT NULL,  
Name varchar(255) NOT NULL,  
Address varchar(255),  
City varchar(255)  
)

**SQL UNIQUE Constraint**

The UNIQUE constraint uniquely identifies each record in a database table.

The UNIQUE and PRIMARY KEY constraints both provide a guarantee for uniqueness for a column or set of columns.

A PRIMARY KEY constraint automatically has a UNIQUE constraint defined on it. There can be many UNIQUE constraints per table, but only one PRIMARY KEY constraint per table.

**SQL UNIQUE Constraint on CREATE TABLE**

The following SQL creates a UNIQUE constraint on the "P\_Id" column when the "Persons" table is created:

CREATE TABLE Persons  
(  
P\_Id int NOT NULL UNIQUE,  
LastName varchar(255) NOT NULL,  
FirstName varchar(255),  
Address varchar(255),  
City varchar(255)  
)

**SQL UNIQUE Constraint on ALTER TABLE**

To create a UNIQUE constraint on the "P\_Id" column when the table is already created, use the following SQL:

ALTER TABLE Persons  
ADD UNIQUE (P\_Id)

To allow naming of a UNIQUE constraint, and for defining a UNIQUE constraint on multiple columns, use the following SQL syntax:

ALTER TABLE Persons  
ADD CONSTRAINT uc\_PersonID UNIQUE (P\_Id,LastName)

**To DROP a UNIQUE Constraint**

To drop a UNIQUE constraint, use the following SQL:

ALTER TABLE Persons  
DROP CONSTRAINT uc\_PersonID

**SQL DEFAULT Constraint**

The DEFAULT constraint is used to insert a default value into a column.

The default value will be added to all new records, if no other value is specified.

**SQL DEFAULT Constraint on CREATE TABLE**

The following SQL creates a DEFAULT constraint on the "City" column when the "Persons" table is created:

CREATE TABLE Persons  
(  
P\_Id int NOT NULL,  
LastName varchar(255) NOT NULL,  
FirstName varchar(255),  
Address varchar(255),  
City varchar(255) DEFAULT 'Sandnes'  
)

The DEFAULT constraint can also be used to insert system values, by using functions like GETDATE():

CREATE TABLE Orders  
(  
O\_Id int NOT NULL,  
OrderNo int NOT NULL,  
P\_Id int,  
OrderDate date DEFAULT GETDATE()  
)

**SQL DEFAULT Constraint on ALTER TABLE**

To create a DEFAULT constraint on the "City" column when the table is already created, use the following SQL:

ALTER TABLE Persons  
MODIFY City DEFAULT 'SANDNES'

**To DROP a DEFAULT Constraint**

To drop a DEFAULT constraint, use the following SQL:

ALTER TABLE Persons  
ALTER COLUMN City DROP DEFAULT

Suppose in the following table we want to make column gpa as default with value 3.0

SQL> desc student1;

Name Null? Type

----------------------------------------- -------- ----------------------

SID NOT NULL CHAR(20)

NAME VARCHAR2(20)

LOGIN CHAR(20)

AGE NUMBER(38)

GPA FLOAT(63)

SQL> alter table student1 modify gpa default 3.0;

Table altered.

Now to add insert data we have to add only values for the columns without default values.

SQL> insert into student1 (sid,name,login,age) values ('106','krish','krish@maths',19);

SQL> select \* from student1;

SID NAME LOGIN AGE GPA

------- ---------- --------------- ---------- ----------

100 xxx xxx1 20 4.5

101 jones jones@cs 18 3.4

102 smith smith@ee 19 3.2

103 madayan madayan@music 11 1.8

106 krish krish@maths 19 3

1 row created.

In the above output data column gpa for sid=106 is not entered in insert command but it is automatically entered.

SQL SELECT Statement:

The SELECT statement is used to select data from a database.

|  |
| --- |
| SELECT column1, column2....columnN  FROM table\_name; |

|  |
| --- |
| SELECT \* FROM table\_name; |

Example

SQL> select ename,street,city,eid from employee\_csc;

ENAME STREET CITY EID

---------- --------------- --------------- ----------

anitha 1st street chennai 100

aiswarya 2nd street chennai 101

chandra 2nd street chennai 102

hema 3rd street chennai 103

lalitha metha street mumbai 104

raman krishnan street bangalore 105

harini kalam street andhra 106

danush ragav street bangalore 107

8 rows selected.

SQL DISTINCT Clause:

In a table, a column may contain many duplicate values; and sometimes you only want to list the different (distinct) values.

The DISTINCT keyword can be used to return only distinct (different) values.

|  |
| --- |
| SELECT DISTINCT column1, column2....columnN  FROM table\_name; |

Example:

SQL> select distinct city from employee\_csc;

CITY

---------------

chennai

mumbai

andhra

bangalore

SQL WHERE Clause:

The WHERE clause is used to extract only those records that fulfill a specified criterion.

|  |
| --- |
| SELECT column1, column2....columnN  FROM table\_name  WHERE CONDITION; |

Example:

SQL> select ename,street,city,eid from employee\_csc

where city='chennai';

ENAME STREET CITY EID

---------- --------------- --------------- ----------

anitha 1st street chennai 100

aiswarya 2nd street chennai 101

chandra 2nd street chennai 102

hema 3rd street chennai 103

Example:

SQL> select \* from employee\_csc where eid=101;

ENAME STREET CITY EID

---------- --------------- --------------- ----------

aiswarya 2nd street chennai 101

SQL AND/OR Clause:

The AND & OR operators are used to filter records based on more than one condition.

The AND operator displays a record if both the first condition AND the second condition are true.

The OR operator displays a record if either the first condition OR the second condition is true.

|  |
| --- |
| SELECT column1, column2....columnN  FROM table\_name  WHERE CONDITION-1 {AND|OR} CONDITION-2; |

**Using And**

SQL> select \* from employee\_csc where city='chennai' and eid=103;

ENAME STREET CITY EID

---------- --------------- --------------- ----------

hema 3rd street chennai 103

**Using Or**

SQL> select \* from employee\_csc where city='chennai' or city='bangalore';

ENAME STREET CITY EID

---------- --------------- --------------- ----------

anitha 1st street chennai 100

aiswarya 2nd street chennai 101

chandra 2nd street chennai 102

hema 3rd street chennai 103

raman krishnan street bangalore 105

danush ragav street bangalore 107

6 rows selected.

SQL IN Clause:

The IN operator allows you to specify multiple values in a WHERE clause.

|  |
| --- |
| SELECT column1, column2....columnN  FROM table\_name  WHERE column\_name IN (val-1, val-2,...val-N); |

SQL> select \* from employee\_csc

where city in('mumbai','chennai');

ENAME STREET CITY EID

---------- --------------- --------------- ----------

anitha 1st street chennai 100

aiswarya 2nd street chennai 101

chandra 2nd street chennai 102

hema 3rd street chennai 103

lalitha metha street mumbai 104

SQL ORDER BY Clause:

The ORDER BY keyword sorts the records in ascending order by default. To sort the records in a descending order, you can use the DESC keyword.

|  |
| --- |
| SELECT column1, column2....columnN  FROM table\_name  ORDER BY column\_name {ASC|DESC}; |

Example:

SQL> select \* from employee\_csc order by city;

ENAME STREET CITY EID

---------- --------------- --------------- ----------

harini kalam street andhra 106

raman krishnan street bangalore 105

danush ragav street bangalore 107

chandra 2nd street chennai 102

aiswarya 2nd street chennai 101

anitha 1st street chennai 100

hema 3rd street chennai 103

lalitha metha street mumbai 104

8 rows selected.

Example: applying order by for more than one column:

SQL> select \* from employee\_csc order by city,ename;

ENAME STREET CITY EID

---------- --------------- --------------- ----------

harini kalam street andhra 106

danush ragav street bangalore 107

raman krishnan street bangalore 105

aiswarya 2nd street chennai 101

anitha 1st street chennai 100

chandra 2nd street chennai 102

hema 3rd street chennai 103

lalitha metha street mumbai 104

8 rows selected.

Example : using descending order (desc)

In this city is in decending order and within that the ename is in ascending order.

SQL> select \* from employee\_csc order by city desc,ename;

ENAME STREET CITY EID

---------- --------------- --------------- ----------

lalitha metha street mumbai 104

aiswarya 2nd street chennai 101

anitha 1st street chennai 100

chandra 2nd street chennai 102

hema 3rd street chennai 103

danush ragav street bangalore 107

raman krishnan street bangalore 105

harini kalam street andhra 106

8 rows selected.

SQL BETWEEN Clause:

The BETWEEN operator selects values within a range. The values can be numbers, text, or dates.

|  |
| --- |
| SELECT *column1, column2....columnN*  FROM *table\_name*  WHERE *column\_name* BETWEEN val-1 AND val-2; |

SQL> select \* from employee\_csc where eid between 100 and 103;

ENAME STREET CITY EID

---------- --------------- --------------- ----------

anitha 1st street chennai 100

aiswarya 2nd street chennai 101

chandra 2nd street chennai 102

hema 3rd street chennai 103

NOT BETWEEN Operator Example

To display the products outside the range of the previous example, use NOT BETWEEN:

SQL> select \* from employee\_csc where eid not between 100 and 103;

ENAME STREET CITY EID

---------- --------------- --------------- ----------

lalitha metha street mumbai 104

raman krishnan street bangalore 105

harini kalam street andhra 106

danush ragav street bangalore 107

Between operator for text

SQL> select \* from employee\_csc where ename between 'a' and 'h';

ENAME STREET CITY EID

---------- --------------- --------------- ----------

anitha 1st street chennai 100

aiswarya 2nd street chennai 101

chandra 2nd street chennai 102

danush ragav street bangalore 107

SQL LIKE Clause:

The LIKE operator is used in a WHERE clause to search for a specified pattern in a column.

|  |
| --- |
| SELECT *column1, column2....columnN*  FROM *table\_name*  WHERE *column\_name* LIKE { PATTERN }; |

Example : This selects record with ename starting with ‘h’

SQL> select \* from employee\_csc where ename like 'h%';

ENAME STREET CITY EID

---------- --------------- --------------- ----------

hema 3rd street chennai 103

harini kalam street andhra 106

Example: this selects records with city has spelling ‘an’ anywhere in its word.

SQL> select \* from employee\_csc where city like '%an%';

ENAME STREET CITY EID

---------- --------------- --------------- ----------

raman krishnan street bangalore 105

harini kalam street andhra 106

danush ragav street bangalore 107

Example: this selects records with city does not have spelling ‘an’ anywhere in its word.

SQL> select \* from employee\_csc where city not like '%an%';

ENAME STREET CITY EID

---------- --------------- --------------- ----------

anitha 1st street chennai 100

aiswarya 2nd street chennai 101

chandra 2nd street chennai 102

hema 3rd street chennai 103

lalitha metha street mumbai 104

Alter Table: altering an existing table

Adding a new column to an existing table

Syntax:

|  |
| --- |
| ALTER TABLE *tablename*  ADD *columnname datatype;* |

Example

Suppose to the existing employee\_csc zip code has to be added can be done as follows:

SQL> alter table employee\_csc add zip numeric(6);

Table altered.

Modify an existing column:

|  |
| --- |
| ALTER TABLE *tablename MODIFYcolumnname newdatatype;* |

suppose we want to modify the datatype of zip to some other it can be done as follows:

SQL> alter table employee\_csc modify zip varchar2(8);

Table altered.

Adding a Constraint using alter table

|  |
| --- |
| *ALTER TABLE tablename*  *ADD [CONSTRAINT constraint\_name] constraint\_type (column,..)* |

Example Adding the primary key constraint to employee\_csc table:

SQL> alter table employee\_csc

add constraint employee\_eid\_pk PRIMARY KEY(eid)

Adding a foreign key:

Suppose we have another table named “manager\_csc” and want to make eid in this table as foreign key, it can be done as follows:

SQL> desc manager\_csc;

Name Null? Type

----------------------------------------- -------- ----------------------------

MNAME VARCHAR2(30)

ENAME VARCHAR2(30)

MID NOT NULL VARCHAR2(3)

EID NUMBER(3)

CID VARCHAR2(4)

SQL> alter table manager\_csc

add constraint manager\_eid\_fk FOREIGN KEY(eid)REFERENCES employee(eid)

Suppose we want to add check constraint to the following table on salary column so that it should not have negative and 0 value, we can do it by:

SQL> desc works\_csc;

Name Null? Type

----------------------------------------- -------- ----------------------------

CNAME VARCHAR2(20)

SALARY NUMBER(7,2)

EID NOT NULL NUMBER(3)

CID VARCHAR2(4)

SQL> alter table works\_csc add constraint works\_salary\_ck

check(salary>0);

Table altered.

Drop column

A column can be deleted or dropped using alter table.

|  |
| --- |
| ALTER TABLE tablename DROP COLUMN columnname; |

Example : in this the zip column is dropped:

SQL> alter table employee\_csc drop column zip;

Table altered.

Renaming a column name

|  |
| --- |
| ALTER TABLE tablename RENAME COLUMN oldname TO newname; |

Example: in this the column name ‘city’ is changed to ‘ccity’

SQL> alter table company\_csc rename column city to ccity;

Table altered.

Dropping a table

A table can be dropped when not needed or found to be fault. When a table is dropped, all data and the table structure are permanently deleted. The drop operation cannot be reversed.

Syntax

|  |
| --- |
| DROP TABLE tablename; |

DROP TABLE student;

Renaming a Table:

A table can be renamed only by its owner.

|  |
| --- |
| RENAME oldtablename to newtablename; |

Truncating a table:

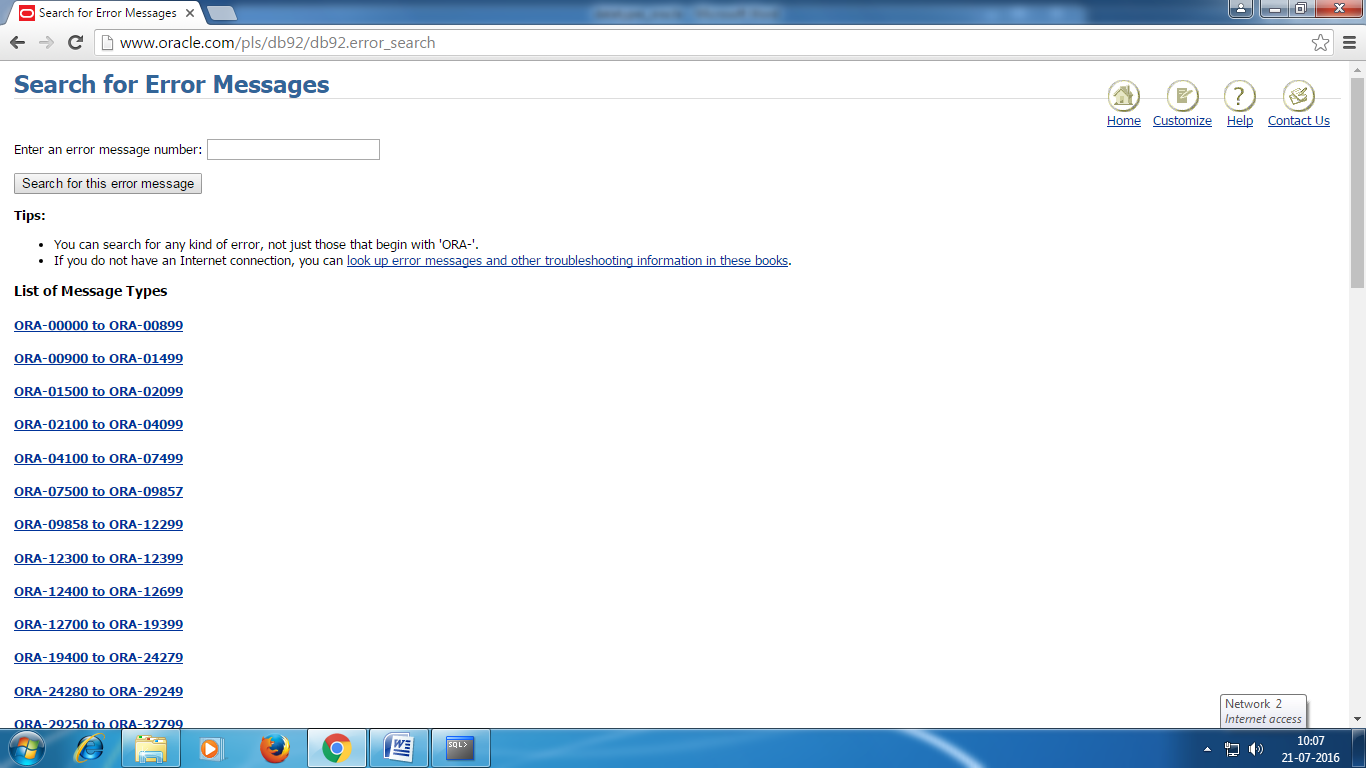
Truncating a table removes only data, the structure of table remains intact.

|  |
| --- |
| TRUNCATE TABLE tablename; |

Error codes

You can go to the following website and type your error code in search box to know the meaning of error.

<http://otn.oracle.com/pls/db92/db92.error_search>



SQL has many built-in functions for performing calculations on data.

**SQL Aggregate Functions**

SQL aggregate functions return a single value, calculated from values in a column.

Useful aggregate functions:

* AVG() - Returns the average value
* COUNT() - Returns the number of rows
* FIRST() - Returns the first value
* LAST() - Returns the last value
* MAX() - Returns the largest value
* MIN() - Returns the smallest value
* SUM() - Returns the sum

SQL Scalar functions

SQL scalar functions return a single value, based on the input value.

Useful scalar functions:

* UCASE() - Converts a field to upper case
* LCASE() - Converts a field to lower case
* MID() - Extract characters from a text field
* LEN() - Returns the length of a text field
* ROUND() - Rounds a numeric field to the number of decimals specified
* NOW() - Returns the current system date and time
* FORMAT() - Formats how a field is to be displayed

SQL COUNT Clause:

The COUNT(column\_name) function returns the number of values (NULL values will not be counted) of the specified column:

|  |
| --- |
| SELECT COUNT(column\_name)  FROM table\_name  WHERE CONDITION; |

The COUNT(\*) function returns the number of records in a table:

SELECT COUNT(\*) FROM table\_name;

Example:

SQL> select count(8) from employee\_csc;

COUNT(8)

----------

8

### SQL COUNT(DISTINCT column\_name) Syntax

The COUNT(DISTINCT column\_name) function returns the number of distinct values of the specified column:

|  |
| --- |
| SELECT COUNT(DISTINCT column\_name) FROM table\_name; |

Example: to know how many city in the table:

SQL> select count(distinct city) from employee\_csc;

COUNT(DISTINCTCITY)

-------------------

4

## The AVG() Function

The AVG() function returns the average value of a numeric column.

### SQL AVG() Syntax

|  |
| --- |
| SELECT AVG(column\_name) FROM table\_name |

Let us take the table ‘works\_csc’ as an example

SQL> select \* from works\_csc;

CNAME SALARY EID CID

-------------------- ---------- ---------- ----

infosys 45000 100 c1

wipro 35000 101 c2

cts 35000 102 c3

tcs 50000 103 c4

wipro 30000 104 c2

cts 30000 105 c3

infosys 40000 106 c1

7 rows selected.

To find the average salary type the following command:

SQL> select avg(salary) from works\_csc;

AVG(SALARY)

-----------

37857.1429

Example :select employeeid(eid) whose salary is more than average salary in works\_csc table

SQL> select eid from works\_csc where salary>(select avg(salary) from works\_csc);

EID

----------

100

103

106

SQL> select avg(salary) from works\_csc where cname='cts';

AVG(SALARY)

-----------

32500

Example:select employee id whose salary is greater than average salary of a particular company.

SQL> select eid from works\_csc where salary>(select avg(salary) from works\_csc where cname='cts');

EID

----------

100

101

102

103

106

SQL>select x.eid,x.ename from employee\_csc x,works\_csc y where salary>(select avg(salary) from works\_csc where cname='cts') and x.eid=y.eid

EID ENAME

---------- ----------

100 anitha

101 aiswarya

102 chandra

103 hema

106 harini

Max function

SQL> select max(salary) from works\_csc;

MAX(SALARY)

-----------

50000

SQL> select x.ename,x.eid,y.salary from employee\_csc x, works\_csc y where y.salary>=(select max(salary) from works\_csc) and x.eid=y.eid;

ENAME EID SALARY

---------- ---------- ----------

hema 103 50000

**GROUPBY CLAUSE**

SQL> select city,count(eid) from employee\_csc

2 group by city;

CITY COUNT(EID)

------------------------------ ----------

chennai 4

mumbai 1

andhra 1

bangalore 2

calcutta 1

**Having clause**

SQL> select city,count(eid) from employee\_csc

2 group by city

3 having count(eid)>=2;

CITY COUNT(EID)

------------------------------ ----------

chennai 4

bangalore 2

SQL Joins

* **INNER JOIN**: Returns all rows when there is at least one match in BOTH tables
* **LEFT JOIN**: Return all rows from the left table, and the matched rows from the right table
* **RIGHT JOIN**: Return all rows from the right table, and the matched rows from the left table
* **FULL JOIN**: Return all rows when there is a match in ONE of the tables

Employee\_csc

ENAME STREET CITY EID

---------- --------------- ---------- ----------

anitha 1st street chennai 100

aiswarya 2nd street chennai 101

chandra 2nd street chennai 102

hema 3rd street chennai 103

lalitha metha street mumbai 104

raman krishnan street bangalore 105

harini kalam street andhra 106

danush ragav street bangalore 107

david kamaraj street calcutta 108

ananthi rajaji street chennai 109

works\_csc

SALARY EID CID

---------- ---------- ----

45000 100 c1

35000 101 c2

35000 102 c3

50000 103 c4

30000 104 c2

30000 105 c3

40000 106 c1

30000 108 c3

28000 109 c3

inner join

SQL> select \* from employee\_csc x inner join works\_csc y

2 on x.eid=y.eid

3 order by x.ename;

EID EMPNAME STREET CITY EID CID SALARY

---------- ---------- ---------- ---------- ---------- ---- ----------

100 anitha 1st street calcutta 100 c1 45000

101 aiswarya 2nd street chennai 101 c2 35000

102 chandra 2nd street chennai 102 c3 35000

103 hema 3rd street chennai 103 c4 50000

104 lalitha metha stre mumbai 104 c2 30000

et

105 raman krishnan s bangalore 105 c3 30000

treet

106 harini kalam stre andhra 106 c1 40000

EID EMPNAME STREET CITY EID CID SALARY

---------- ---------- ---------- ---------- ---------- ---- ----------

et

108 david kamaraj st calcutta 108 c3 30000

reet

109 ananthi rajaji str chennai 109 c3 28000

eet

9 rows selected.

The INNER JOIN keyword selects all rows from both tables as long as there is a match between the columns. If there are rows in the "employee\_csc" table that do not have matches in "works\_csc", these customers will NOT be listed.

Manager\_csc data

MNAME MID EID CID

------------------------------ --- ---------- ----

ajith m1 100 c1

hari m2 105 c3

karthik m3 104 c2

janani m4 101 c2

krishnan m5 103 c4

jothi m6 102 c3

dhanush m7 107 c4

SQL>select \* from employee\_csc x inner join manager\_cscy

on x.eid=y.eid

order by x.eid

## EID EMPNAME STREET CITY MID MNAME EID CID

## ----- ---------- --------------- ---------- --- ---------- ----- ----

## 100 anitha 1st street calcutta m1 ajith 100 c1

## 101 aiswarya 2nd street chennai m4 janani 101 c2

## 102 chandra 2nd street chennai m6 jothi 102 c3

## 103 hema 3rd street chennai m5 krishnan 103 c4

## 104 lalitha metha street mumbai m3 karthik 104 c2

## 105 raman krishnan street bangalore m2 hari 105 c3

## 107 danush ragav street bangalore m7 dhanush 107 c4

## 7 rows selected.

## SQL LEFT JOIN Keyword

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.The LEFT JOIN keyword returns all the rows from the left table (employee\_csc), even if there are no matches in the right table (manager\_csc).

### SQL LEFT JOIN Syntax

|  |
| --- |
| SELECT column\_name(s) FROM table1 LEFTJOIN table2 ON table1.column\_name=table2.column\_name; |

SQL>select \* from employee\_csc x left join manager\_csc y

on x.eid=y.eid

order by x.eid

EID EMPNAME STREET CITY MID MNAME EID CID

----- ---------- --------------- ---------- --- ---------- ----- ----

100 anitha 1st street calcutta m1 ajith 100 c1

101 aiswarya 2nd street chennai m4 janani 101 c2

102 chandra 2nd street chennai m6 jothi 102 c3

103 hema 3rd street chennai m5 krishnan 103 c4

104 lalitha metha street mumbai m3 karthik 104 c2

105 raman krishnan street bangalore m2 hari 105 c3

106 harini kalam street andhra

107 danush ragav street bangalore m7 dhanush 107 c4

108 david kamaraj street calcutta

109 ananthi rajaji street chennai

112 krish 3rd street bangalore

11 rows selected.

select\* from manager\_csc x left join employee\_csc y

on x.eid=y.eid

order by x.eid

MID MNAME EID CID EID EMPNAME STREET CITY

--- ---------- ----- ---- ----- ---------- --------------- ----------

m1 ajith 100 c1 100 anitha 1st street calcutta

m4 janani 101 c2 101 aiswarya 2nd street chennai

m6 jothi 102 c3 102 chandra 2nd street chennai

m5 krishnan 103 c4 103 hema 3rd street chennai

m3 karthik 104 c2 104 lalitha metha street mumbai

m2 hari 105 c3 105 raman krishnan street bangalore

m7 dhanush 107 c4 107 danush ragav street bangalore

7 rows selected.

## SQL RIGHT JOIN Keyword

The RIGHT JOIN keyword returns all rows from the right table (table2), with the matching rows in the left table (table1). The result is NULL in the left side when there is no match. The RIGHT JOIN keyword returns all the rows from the right table (manager\_csc), even if there are no matches in the left table (employee\_csc).

### SQL RIGHT JOIN Syntax

SELECT column\_name(s)  
FROM table1  
RIGHT JOIN table2  
ON table1.column\_name=table2.column\_name;

SQL>select \* from employee\_csc x right join manager\_csc y

on x.eid=y.eid

order by x.eid

EID EMPNAME STREET CITY MID MNAME EID CID

----- ---------- --------------- ---------- --- ---------- ----- ----

100 anitha 1st street calcutta m1 ajith 100 c1

101 aiswarya 2nd street chennai m4 janani 101 c2

102 chandra 2nd street chennai m6 jothi 102 c3

103 hema 3rd street chennai m5 krishnan 103 c4

104 lalitha metha street mumbai m3 karthik 104 c2

105 raman krishnan street bangalore m2 hari 105 c3

107 danush ragav street bangalore m7 dhanush 107 c4

7 rows selected.

select\* from manager\_csc x right join employee\_csc y

on x.eid=y.eid

order by x.eid

## MID MNAME EID CID EID EMPNAME STREET CITY

## --- ---------- ----- ---- ----- ---------- --------------- ----------

## m1 ajith 100 c1 100 anitha 1st street calcutta

## m4 janani 101 c2 101 aiswarya 2nd street chennai

## m6 jothi 102 c3 102 chandra 2nd street chennai

## m5 krishnan 103 c4 103 hema 3rd street chennai

## m3 karthik 104 c2 104 lalitha metha street mumbai

## m2 hari 105 c3 105 raman krishnan street bangalore

## m7 dhanush 107 c4 107 danush ragav street bangalore

## 108 david kamaraj street calcutta

## 112 krish 3rd street bangalore

## 109 ananthi rajaji street chennai

## 106 harini kalam street andhra

## 11 rows selected.

## SQL FULL OUTER JOIN Keyword

The FULL OUTER JOIN keyword returns all rows from the left table (table1) and from the right table (table2).

The FULL OUTER JOIN keyword combines the result of both LEFT and RIGHT joins.

### SQL FULL OUTER JOIN Syntax

SELECT column\_name(s)  
FROM table1  
FULL OUTER JOIN table2  
ON table1.column\_name=table2.column\_name;

select\* from employee\_csc x full outer join manager\_csc y on x.eid=y.eid

order by x.eid

EID EMPNAME STREET CITY MID MNAME EID CID

----- ---------- --------------- ---------- --- ---------- ----- ----

100 anitha 1st street calcutta m1 ajith 100 c1

101 aiswarya 2nd street chennai m4 janani 101 c2

102 chandra 2nd street chennai m6 jothi 102 c3

103 hema 3rd street chennai m5 krishnan 103 c4

104 lalitha metha street mumbai m3 karthik 104 c2

105 raman krishnan street bangalore m2 hari 105 c3

106 harini kalam street andhra

107 danush ragav street bangalore m7 dhanush 107 c4

108 david kamaraj street calcutta

109 ananthi rajaji street chennai

112 krish 3rd street bangalore

11 rows selected.

The FULL OUTER JOIN keyword returns all the rows from the left table (employee\_csc), and all the rows from the right table (manager\_csc). If there are rows in "employee\_csc" that do not have matches in "manager\_csc", or if there are rows in "manager\_csc" that do not have matches in "employee\_csc ", those rows will be listed as well.

Set operators

**UNION :**

The UNION set operator returns all distinct rows selected by either query. That means any duplicate rows will be removed.

SQL> select \* from employee\_csc where city='chennai'

union

select \* from employee\_csc where city='bangalore';

ENAME STREET CITY EID

--------------- --------------- --------------- ----------

aiswarya 2nd street chennai 101

anitha 1st street chennai 100

chandra 2nd street chennai 102

danush ragav street bangalore 107

hema 3rd street chennai 103

raman krishnan street bangalore 105

6 rows selected.

select \* from employee\_csc where city='chennai'

union

select \* from employee\_csc where eid>107

ENAME STREET CITY EID

--------------- --------------- --------------- ----------

aiswarya 2nd street chennai 101

anitha 1st street chennai 100

chandra 2nd street chennai 102

david kamaraj street calcutta 108

hema 3rd street chennai 103

another example

select eid, empname from employee where city='chennai' union select eid, mname from manager\_csc

SQL> /

EID EMPNAME

---------- ------------------------------

100 ajith

100 anitha

101 aiswarya

101 janani

102 chandra

102 jothi

103 hema

103 krishnan

104 karthik

105 hari

107 dhanush

EID EMPNAME

---------- ------------------------------

109 ananthi

12 rows selected.

**INTERSECT**

The INTERSECT set operator returns all distinct rows selected by both queries. That means only those rows common to both queries will be present in the final result set.

select \* from employee\_csc where eid>=103

intersect

select \* from employee\_csc where eid<107

ENAME STREET CITY EID

--------------- --------------- --------------- ----------

harini kalam street andhra 106

hema 3rd street chennai 103

lalitha metha street mumbai 104

raman krishnan street bangalore 105

Example 2:

1\* select eid from employee where eid>=100 intersect select eid from manager\_csc where eid<=107

SQL> /

EID

----------

100

101

102

103

104

105

107

7 rows selected.

MINUS

The MINUS set operator returns all distinct rows selected by the first query but not the second. This is functionally equivalent to the ANSI set operator EXCEPT DISTINCT.

1 select \* from employee\_csc where eid>=103

2 minus

3\* select \* from employee\_csc where eid<107

SQL> /

ENAME STREET CITY EID

--------------- --------------- --------------- ----------

danush ragav street bangalore 107

david kamaraj street calcutta 108

sql>select eid, empname from employee minus select eid, mname from manager\_csc

/

EID EMPNAME

----- ----------

100 anitha

101 aiswarya

102 chandra

103 hema

104 lalitha

105 raman

106 harini

107 danush

108 david

109 ananthi

112 krish

11 rows selected.

sql>select eid,mname from manager\_csc minus select eid,empname from employee

/

EID MNAME

----- ----------

100 ajith

101 janani

102 jothi

103 krishnan

104 karthik

105 hari

107 dhanush

7 rows selected.

ORDER BY

without order by clause

select \* from employee\_csc where eid>=103

union

select \* from employee\_csc where eid<107

SQL> /

ENAME STREET CITY EID

--------------- --------------- --------------- ----------

aiswarya 2nd street chennai 101

anitha 1st street chennai 100

chandra 2nd street chennai 102

danush ragav street bangalore 107

david kamaraj street calcutta 108

harini kalam street andhra 106

hema 3rd street chennai 103

lalitha metha street mumbai 104

raman krishnan street bangalore 105

9 rows selected.

select \* from employee\_csc where eid>=103

union

select \* from employee\_csc where eid<107

order by 3

SQL> /

ENAME STREET CITY EID

--------------- --------------- --------------- ----------

harini kalam street andhra 106

danush ragav street bangalore 107

raman krishnan street bangalore 105

david kamaraj street calcutta 108

aiswarya 2nd street chennai 101

anitha 1st street chennai 100

chandra 2nd street chennai 102

hema 3rd street chennai 103

lalitha metha street mumbai 104

9 rows selected.

Defining User Variables:

To define a user variable NAME and give it the value “MALA” enter the following command

SQL> DEFINE NAME=MALA

NOTE: Any user variable defines through DEFINE is considered to be of CHAR data type.

To see what has been assigned type the following

SQL>DEFINE NAME

SQL\*Plus lists the definition as:

DEFINE NAME = “MALA” (CHAR)

To delete the definition use UNDEFINE followed by the variable name

Use of Substitution variables:

A substitution variable is a user variable name preceded by one or two ampersands (&). When SQL\*Plus encounters a substitution variable in a command, SQL\*Plus executes the command as though it contained the value of the substitution variable, rather than the variable itself.

SQL\*Plus reads user response from the keyboard and substitutes the value for that variable.

In the below example the insert command has four variables. The table employee consists of four field. The task is to insert values into these fields. To do that the field values are taken as input through the variables. In this example &eid, &empname, &street, &city are used as variables for the fields eid, empname, street and city respectively. Now the SQL prompt the user by adding “Enter the value ‘variablename’: “ .So the user is able to track to which field he is entering the data.

When the user enter the data the values are substituted in the variables for the field names and the command is executed. This is clearly seen in the following command. SQL displays the old command with substitution variables and new command with the values substituted in its place.

SQL> insert into employee values(&eid,'&empname','&street','&city');

Enter value for eid: 100

Enter value for empname: anitha

Enter value for street: 1st street

Enter value for city: chennai

old 1: insert into employee values(&eid,'&empname','&street','&city')

new 1: insert into employee values(100,'anitha','1st street','chennai')

1 row created.

Now if the user wants to execute the insert command with different values it is simple. Just execute the command by the symbol ‘ \ ’. The command get executed again like this:

SQL> /

Enter value for eid: 101

Enter value for empname: aiswary

Enter value for street: 2ND STREET

Enter value for city: chennai

old 1: insert into employee values(&eid,'&empname','&street','&city')

new 1: insert into employee values(101,'aiswary','2ND STREET','chennai')

1 row created.

To display the list of all existing tables in data base

**SQL> select table\_name from user\_tables;**

## What is PL/SQL?

**PL/SQL** stands for **Procedural Language** extension of SQL.

PL/SQL is a combination of SQL along with the procedural features of programming languages.

It was developed by Oracle Corporation in the early 90’s to enhance the capabilities of SQL.

## Architecture of PL/SQL

The PL/SQL architecture mainly consists of following 3 components:

1. PL/SQL block
2. PL/SQL Engine
3. Database Server

### PL/SQL block:

* This is the component which has the actual PL/SQL code.
* This consists of different sections to divide the code logically (declarative section for declaring purpose, execution section for processing statements, exception handling section for handling errors)
* It also contains the [SQL](https://www.guru99.com/sql.html) instruction that used to interact with the database server.
* All the PL/SQL units are treated as PL/SQL blocks, and this is the starting stage of the architecture which serves as the primary input.
* Following are the different type of PL/SQL units.
  + Anonymous Block
  + Function
  + Library
  + Procedure
  + Package Body
  + Package Specification
  + Trigger
  + Type
  + Type Body

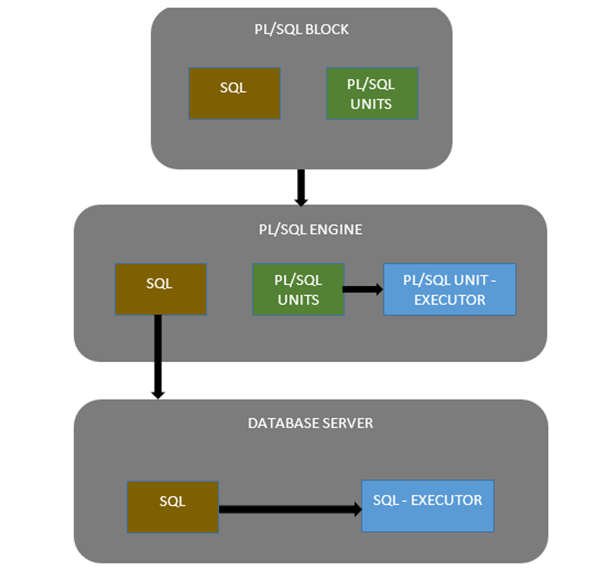
### PL/SQL Engine

* PL/SQL engine is the component where the actual processing of the codes takes place.
* PL/SQL engine separates PL/SQL units and [SQL](https://www.guru99.com/sql.html) part in the input (as shown in the image below).
* The separated PL/SQL units will be handled with the PL/SQL engine itself.
* The [SQL](https://www.guru99.com/sql.html) part will be sent to database server where the actual interaction with database takes place.
* It can be installed in both database server and in the application server.

**Database Server:**

* This is the most important component of Pl/SQL unit which stores the data.
* The PL/SQL engine uses the SQL from PL/SQL units to interact with the database server.
* It consists of SQL executor which actually parses the input SQL statements and execute the same.

Below is the pictorial representation of Architecture of PL/SQL.



## Basic Difference between SQL and PL/SQL

In this section, we will discuss some differences between SQL and PL/SQL

|  |  |
| --- | --- |
| **SQL** | **PL/SQL** |
| * SQL is a single query that is used to perform DML and DDL operations. | * PL/SQL is a block of codes that used to write the entire program blocks/ procedure/ function, etc. |
| * It is declarative, that defines what needs to be done, rather than how things need to be done. | * PL/SQL is procedural that defines how the things needs to be done. |
| * Execute as a single statement. | * Execute as a whole block. |
| * Mainly used to manipulate data. | * Mainly used to create an application. |
| * Interaction with Database server. | * No interaction with the database server. |
| * Cannot contain PL/SQL code in it. | * It is an extension of SQL, so it can contain SQL inside it. |

## Block Structure

PL/SQL blocks have a pre-defined structure in which the code is to be grouped. Below are different sections of PL/SQL blocks

1. Declaration section
2. Execution section
3. Exception-Handling section

### Declaration Section

This is the first section of the PL/SQL blocks. This section is an optional part. This is the section in which the declaration of variables, cursors, exceptions, subprograms, pragma instructions and collections that are needed in the block will be declared. Below are few more characteristics of this part.

* This particular section is optional and can be skipped if no declarations are needed.
* This should be the first section in a PL/SQL block, if present.
* This section starts with the keyword 'DECLARE' for triggers and anonymous block. For other subprograms this keyword will not be present, instead the part after the subprogram name definition marks the declaration section.
* This section should be always followed by execution section.

### Execution Section

Execution part is the main and mandatory part which actually executes the code that is written inside it. Since the PL/SQL expects the executable statements from this block this cannot be an empty block, i.e., it should have at least one valid executable code line in it. Below are few more characteristics of this part.

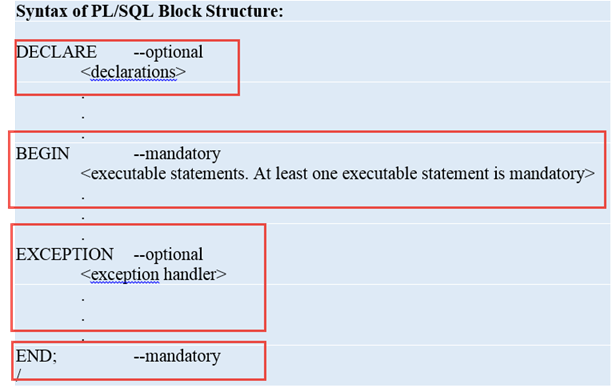
* This can contain both PL/SQL code and [SQL](https://www.guru99.com/sql.html) code.
* This can contain one or many blocks inside it as a nested blocks.
* This section starts with the keyword 'BEGIN'.
* This section should be followed either by 'END' or Exception-Handling section (if present)

### Exception-Handling Section:

The exception are unavoidable in the program which occurs at run-time and to handle this Oracle has provided an Exception-handling section in blocks. This section can also contain PL/SQL statements. This is an optional section of the PL/SQL blocks.

* This is the section where the exception raised in the execution block is handled.
* This section is the last part of the PL/SQL block.
* Control from this section can never return to the execution block.
* This section starts with the keyword 'EXCEPTION'.
* This section should be always followed by the keyword 'END'.

The Keyword 'END' marks the end of PL/SQL block. Below is the syntax of the PL/SQL block structure.

[](https://cdn.guru99.com/images/PL-SQL/110215_0632_BlocksinPLS2.png)

**Note:** A block should be always followed by '/' which sends the information to the compiler about the end of the block.

## Types of PL/SQL block

PL/SQL blocks are of mainly two types.

1. Anonymous blocks
2. Named Blocks

**Anonymous blocks:**

Anonymous blocks are PL/SQL blocks which do not have any names assigned to them. They need to be created and used in the same session because they will not be stored in the server as a database objects.

Since they need not to store in the database, they need no compilation steps. They are written and executed directly, and compilation and execution happen in a single process.

Below are few more characteristics of Anonymous blocks.

* These blocks don't have any reference name specified for them.
* These blocks start with the keyword 'DECLARE' or 'BEGIN'.
* Since these blocks are not having any reference name, these cannot be stored for later purpose. They shall be created and executed in the same session.
* They can call the other named blocks, but call to anonymous block is not possible as it is not having any reference.
* It can have nested block in it which can be named or anonymous. It can also be nested to any blocks.
* These blocks can have all three sections of the block, in which execution section is mandatory, the other two sections are optional.

**Named blocks:**

Named blocks are having a specific and unique name for them. They are stored as the database objects in the server. Since they are available as database objects, they can be referred to or used as long as it is present in the server. The compilation process for named blocks happens separately while creating them as a database objects.

Below are few more characteristics of Named blocks.

* These blocks can be called from other blocks.
* The block structure is same as an anonymous block, except it will never start with the keyword 'DECLARE'. Instead, it will start with the keyword 'CREATE' which instruct the compiler to create it as a database object.
* These blocks can be nested within other blocks. It can also contain nested blocks.
* Named blocks are basically of two types:

1. Procedure
2. Function

## PL/SQL block structure example

Let’s take a look at the simplest PL/SQL block that does nothing.

|  |  |
| --- | --- |
| 1  2  3 | BEGIN     NULL;  END; |

If you execute the above anonymous block in SQL\*Plus you will see that it issues a message

|  |  |
| --- | --- |
| 1 | PL/SQLproceduresuccessfullycompleted. |

Because the NULL statement does nothing.

To display database’s output on the screen, you need to:

* First, use the SET SERVEROUTPUT ON command to instruct SQL\*Plus to echo database’s output after executing the PL/SQL block. The SET SERVEROUTPUT ON is SQL\*Plus command, which is not related to PL/SQL.
* Second, use the DBMS\_OUTPUT.PUT\_LINE procedure to output a string on the screen.

The following example displays a message Hello PL/SQL on a screen using SQL\*Plus:

|  |  |
| --- | --- |
| 1  2  3  4  5 | SETSERVEROUTPUTON  BEGIN     DBMS\_OUTPUT.PUT\_LINE('Hello PL/SQL');  END;  / |

PL/SQL variables

In PL/SQL, a variable is a meaningful name of a temporary storage location that supports a particular data type in a program. A variableneed to be declaredbefore it is used, first in the declaration section of a [PL/SQL block](http://www.plsqltutorial.com/plsql-block-structure/).

## PL/SQL variables naming rules

Like other programming languages, a variable in PL/SQL must follow the naming rules as follows:

* The variable name must be less than 31 characters. Try to make it as meaningful as possible within 31 characters.
* The variable name must begin with an ASCII letter. It can be either lowercase or uppercase and is case-insensitive, which means **data** and **DATA**refer to the same variable.
* Followed by the first character are any number, underscore ( \_), and dollar sign ( $) characters.

PL/SQL variables naming convention : it is better to follow the following naming convention for variables.

| **Prefix** | **Data Type** |
| --- | --- |
| v\_ | VARCHAR2 |
| n\_ | NUMBER |
| t\_ | TABLE |
| r\_ | ROW |
| d\_ | DATE |
| b\_ | BOOLEAN |

For example, if you want to declare a variable that holds the first name of employee with the VARCHAR2 data type, and salary of NUMBER data type then  the variable name should be v\_first\_name and n\_salary respectively.

## CHARACTER Data type:

This data type basically stores alphanumeric characters in string format.

The literal values should always be enclosed between single quotes while assigning them to CHARACTER data type.

This character data type is further classified as follows:

* CHAR Data type (fixed string size)
* VARCHAR2 Data type (variable string size)
* VARCHAR Data type
* NCHAR (native fixed string size)
* NVARCHAR2 (native variable string size)
* LONG and LONG RAW

### CHAR Data type:

This data type stores the string value, and the size of the string is fixed at the time of declaring the variable.

* Oracle will be blank-padded the variable if the variable didn't occupy the entire size that has been declared for it, hence oracle will allocate the memory for declared size even if the variable didn't occupy it fully.
* The size restriction for this data type is 1-2000 bytes.
* CHAR data type is more appropriate to use where ever fixed size of data will be handled. .

[](https://cdn.guru99.com/images/PL-SQL/110215_0714_PLSQLDataTy1.png)

**Syntax Explanation:**

* The first declaration statement declares the variable 'grade' of CHAR data type with the maximum size of 1 byte (default value).
* The second declaration statement declares the variable 'manager' of CHAR data type with the maximum size of 10 and assigned the value 'guru99' which is of 6 bytes. Oracle will allocate the memory of 10 bytes rather than 6 bytes in this case.

### VARCHAR2 Data type:

This data type stores the string, but the length of the string is not fixed.

* The size restriction for this data type is 1-4000 bytes for table column size and 1-32767 bytes for variables.
* The size is defined for each variable at the time of variable declaration.
* But Oracle will allocate memory only after the variable is defined, i.e., Oracle will consider only the actual length of the string that is stored in a variable for memory allocation rather than the size that have been given for a variable in the declaration part.
* It is always good to use VARCHAR2 instead of CHAR data type to optimize the memory usage.

[PL/SQL Data Types](https://cdn.guru99.com/images/PL-SQL/110215_0714_PLSQLDataTy2.png)

**Syntax Explanation:**

* The above declaration statement declares the variable 'manager' of VARCHAR2 data type with the maximum size of 10 and assigned the value 'guru99' which is of 6 bytes. Oracle will allocate memory of only 6 bytes in this case.

### VARCHAR Data type:

This is the synonymous with the VARCHAR2 data type.

* It is always a good practice to use VARCHAR2 instead of VARCHAR to avoid behavioral changes.

[PL/SQL Data Types](https://cdn.guru99.com/images/PL-SQL/110215_0714_PLSQLDataTy3.png)

**Syntax Explanation:**

* The above declaration statement declares the variable 'manager' of VARCHAR data type with the maximum size of 10 and assigned the value 'guru99' which is of 6 bytes. Oracle will allocate memory of only 6 bytes in this case. (Similar to VARCHAR2)

### NCHAR Data type:

This data type is same as CHAR data type, but the character set will of national character set.

* This character set can be defined for the session using NLS\_PARAMETERS.
* The character set can be either UTF16 or UTF8.
* The size restriction is 1-2000 bytes.

[PL/SQL Data Types](https://cdn.guru99.com/images/PL-SQL/110215_0714_PLSQLDataTy4.png)

**Syntax Explanation:**

* The above declaration statement declares the variable 'native' of NCHAR data type with the maximum size of 10.
* The length of this variable depends upon the (number of length) per byte as defined in the character set.

### NVARCHAR2 Data type:

This data type is same as VARCHAR2 data type, but the character set will be of national character set.

* This character set can be defined for the session using NLS\_PARAMETERS.
* The character set can be either UTF16 or UTF8.
* The size restriction is 1-4000 bytes.

[PL/SQL Data Types](https://cdn.guru99.com/images/PL-SQL/110215_0714_PLSQLDataTy5.png)

**Syntax Explanation:**

* The above declaration statement declares the variable 'Native\_var' of NVARCHAR2 data type with the maximum size of 10.

### LONG and LONGRAW Data type:

This data type is used to store large text or raw data up to the maximum size of 2GB.

* These are mainly used in the data dictionary.
* LONG data type is used to store character set data, while LONG RAW is used to store data in binary format.
* LONG RAW data type accept media objects, images, etc. whereas LONG works only on data that can be stored using character set.

[](https://cdn.guru99.com/images/PL-SQL/110215_0714_PLSQLDataTy6.png)

**Syntax Explanation:**

* The above declaration statement declares the variable 'Large\_text' of LONG data type and 'Large\_raw' of LONG RAW data type.

**Note:** Using LONG data type is not recommended by Oracle. Instead LOB data type should be preferred.

## NUMBER Data type:

This data type stores fixed or floating point numbers up to 38 digits of precision. This data type is used to work with fields which will contain only number data. The variable can be declared either with precision and decimal digit details or without these information. Values need not to enclose within quotes while assigning for this data type.

[](https://cdn.guru99.com/images/PL-SQL/110215_0714_PLSQLDataTy7.png)

**Syntax Explanation:**

* In the above, the first declaration declares the variable 'A' is of number data type with total precision 8 and decimal digits 2.
* The second declaration declares the variable 'B' is of number data type with total precision 8 and no decimal digits.
* The third declaration is the most generic, declares variable 'C' is of number data type with no restriction in precision or decimal places. It can take up to a maximum of 38 digits.

## BOOLEAN Data type:

This data type stores the logical values. It represents either TRUE or FALSE and mainly used in conditional statements. Values need not enclose within quotes while assigning for this data type.

[PL/SQL Data Types](https://cdn.guru99.com/images/PL-SQL/110215_0714_PLSQLDataTy8.png)

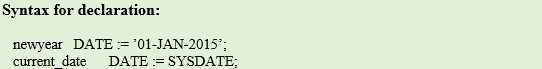
**Syntax Explanation:**

* In the above, variable 'Var1' is declared as BOOLEAN data type. The output of the code will be either true or false based on the condition set.

## DATE Data type:

This data type stores the values in date format, as date, month, and year. Whenever a variable is defined with DATE data type along with the date it can hold time information and by default time information is set to 12:00:00 if not specified. Values need to enclose within quotes while assigning for this data type.

The standard oracle time format for input and output is 'DD-MON-YY' and it is again set at NLS\_PARAMETERS (NLS\_DATE\_FORMAT) at the session level.

[](https://cdn.guru99.com/images/PL-SQL/110215_0714_PLSQLDataTy9.png)

**Syntax Explanation:**

* In the above, variable 'newyear' is declared as DATE data type and assigned the value of Jan 1st, 2015 date.
* The second declaration declares the variable current\_date as DATE data type and assigned the value with current system date.
* Both these variable holds the time information.

## LOB Data type:

This data type is mainly used to store and manipulate large blocks of unstructured data's like **images, multimedia** files, etc. Oracle prefers LOB instead of the LONG data type as it is more flexible than LONG data type. The below are the few main advantage of LOB over LONG data type.

* The number of column in a table with LONG data type is limited to 1, whereas a table has no restriction on number of columns with LOB data type.
* The data interface tool accepts LOB data type of the table during data replication, but it omits LONG column of the table. These LONG columns need to be replicated manually.
* The size of LONG column is 2GB, whereas LOB can store up to 128 TB.
* Oracle is constantly improvising the LOB data type in each of their releases according to the modern requirement, whereas LONG data type is constant and not getting much updates.

So, it is always good to use LOB data type instead of LONG data type. Following are the different LOB data types. They can store up to the size of 128 terabytes.

1. BLOB
2. CLOB and NCLOB
3. BFILE

### BLOB:

This data type stores the LOB data in the binary file format up to the maximum size of 128 TB. This doesn't store data based on the character set details, so it can store the unstructured data such as multimedia objects, images, etc.

[PL/SQL Data Types](https://cdn.guru99.com/images/PL-SQL/110215_0714_PLSQLDataTy10.png)

**Syntax Explanation:**

* In the above, variable 'Binary\_data' is declared as BLOB.

### CLOB and NCLOB:

CLOB data type stores the LOB data into the character set, whereas NCLOB stores the data in the native character set. Since these data types uses character set based storage, these cannot store the data like multimedia, images, etc. that cannot be put into a character string. The maximum size of these data types is 128 TB.

[PL/SQL Data Types](https://cdn.guru99.com/images/PL-SQL/110215_0714_PLSQLDataTy11.png)

**Syntax Explanation:**

* In the above, variable 'Charac\_data' is declared as CLOB data type.

**BFILE:**

* BFILE are the data types that stored the unstructured binary format data outside the database as an operating-system file.
* The size of BFILE is to a limited operating system, and they are read-only files and can't be modified.

## PL/SQL Variables Declaration

To declare a variable, you use a variable name followed by the data type and terminated by a semicolon ( ;). You can also explicitly add a length constraint to the data type within parentheses. The following illustrates some examples of declaring variables in a PL/SQL [anonymous block](http://www.plsqltutorial.com/plsql-block-structure/):

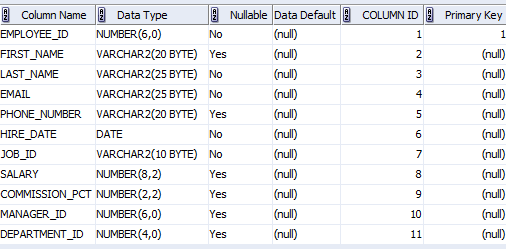
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | DECLARE     v\_first\_namevarchar2(20);     v\_last\_namevarchar2(20);     n\_employee\_idnumber;     d\_hire\_datedate;  BEGIN     NULL;  END; |

## PL/SQL variable anchors

In PL/SQL program, one of the most common tasks is to select values from columns in a table into a set of variables. In case the data types of columns of the table changes, you have to change the PL/SQL program to make the types of the variables compatible with the new changes.

PL/SQL provides you with a very useful feature called **variable anchors**. It refers to the use of  the  %TYPE  keyword to declare a variable with the data type is associated with a column’s data type of a particular column in a table.

Let’s take a look at the employees table in HR sample database provided by Oracle:

[](http://www.plsqltutorial.com/wp-content/uploads/2011/05/Employees_Table.gif)

Employees Table

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | DECLARE    v\_first\_name  EMPLOYEES.FIRST\_NAME%TYPE;    v\_last\_name   EMPLOYEES.LAST\_NAME%TYPE;    n\_employee\_idEMPLOYEES.EMPLOYEE\_ID%TYPE;    d\_hire\_date   EMPLOYEES.HIRE\_DATE%TYPE;  BEGIN    NULL;  END;  / |

The v\_first\_name variable has data type that is the same as the data type of the first\_name column in the  emloyees  table. In case the data type of the first\_name column changes, the  type of the v\_first\_name variable is automatically inherits the new data type of the column.

## PL/SQL variable assignment

In PL/SQL, to assign a value or a variable to a variable, you use the assignment operator ( := ) which is a colon( : ) followed by the equal sign( = ).

Please see the code listing below to get a better understanding:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | DECLARE     v\_first\_nameEMPLOYEES.FIRST\_NAME%TYPE;     v\_last\_nameEMPLOYEES.LAST\_NAME%TYPE;     n\_employee\_idEMPLOYEES.EMPLOYEE\_ID%TYPE;     d\_hire\_dateEMPLOYEES.HIRE\_DATE%TYPE;  BEGIN     v\_first\_name:='Mary';     v\_last\_name:='Jane';     d\_hire\_date:=to\_date('19700101','YYYYMMDD');  END;  / |

In the example above, we assigned Mary to v\_first\_name variable, Jane to v\_last\_name variable, and result of the to\_date function to d\_hire\_date variable.

You can use INTO of the SQL SELECT statement to assign a value to a variable. The INTO clause moves the values from the SELECT query’s column list into corresponding PL/SQL variables.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | SETSERVEROUTPUTONSIZE1000000;  DECLARE     v\_first\_nameEMPLOYEES.FIRST\_NAME%TYPE;     v\_last\_nameEMPLOYEES.LAST\_NAME%TYPE;     n\_employee\_idEMPLOYEES.EMPLOYEE\_ID%TYPE;     d\_hire\_dateEMPLOYEES.HIRE\_DATE%TYPE;  BEGIN     SELECTemployee\_id, first\_name, last\_name, hire\_date   INTOn\_employee\_id,          v\_first\_name,   v\_last\_name,    d\_hire\_date   FROMemployees     WHEREemployee\_id=200;     DBMS\_OUTPUT.PUT\_LINE(v\_first\_name);     DBMS\_OUTPUT.PUT\_LINE(v\_last\_name);     DBMS\_OUTPUT.PUT\_LINE(d\_hire\_date);  END;  / |

## Initializing variables

When you declare a variable, its value is uninitialized and hence is NULL. You can initialize variable a value in declaration section by using variable assignment.

See the following example:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | DECLARE    n\_employee\_idEMPLOYEES.EMPLOYEE\_ID%TYPE:=200;    d\_hire\_dateEMPLOYEES.HIRE\_DATE%TYPE:=to\_date('19700101','YYYYMMDD');  BEGIN     NULL;  END;  / |

In PL/SQL, NULL means an unknown value so it has some special characteristics as follows:

* NULL is not equal to anything, even itself NULL.
* NULL is not greater than or less than anything else, even NULL.
* You cannot use logical operator equal ( =) or ( <>) with NULL. You must use the SQL [IS NULL](http://www.zentut.com/sql-tutorial/sql-is/) or [IS NOT NULL](http://www.zentut.com/sql-tutorial/sql-is/) to test the NULL values.

## Commenting Codes

Commenting code simply instructs the compiler to ignore that particular code from executing.

Comment can be used in the program to increase the readability of the program. In PL/SQL codes can be commented in two ways

* Using '--' in the beginning of the line to comment that particular line.
* Using '/\*…….\*/' we can multiple lines. The symbol '/\*' marks the starting of the comment and the symbol '\*/' marks the end of the comment. The code between these two symbols will be treated as comments by the compiler.

**Example**: In this example, we are going to print 'Hello World' and we are also going to see how the commented lines behave in the code

[](https://cdn.guru99.com/images/PL-SQL/110215_0652_PLSQLFirstP6.png)

## Example of initilizing variable

Let's take a simple example to explain it well:

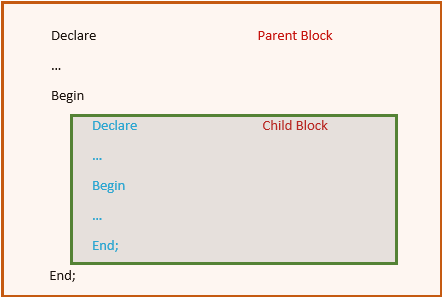
1. DECLARE
2. a integer := 30;
3. b integer := 40;
4. c integer;
5. f real;
6. BEGIN
7. c := a + b;
8. dbms\_output.put\_line('Value of c: ' || c);
9. f := 100.0/3.0;
10. dbms\_output.put\_line('Value of f: ' || f);
11. END;

After the execution, this will produce the following result:

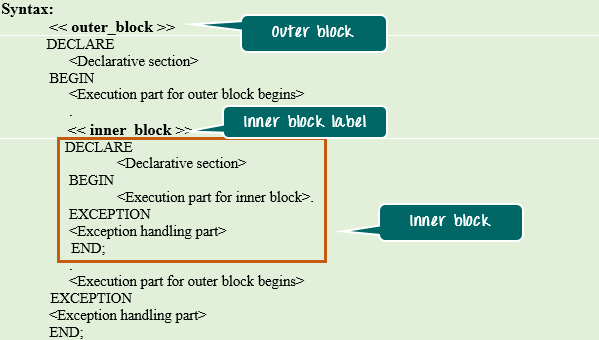
1. Value of c: 70
2. Value of f: 33.333333333333333333
4. PL/SQL procedure successfully completed.

## Nested Block Structure

A block can be nested into another block. This can be nested either in the execution part or in the exception handling part. These block can also be labelled. One outer block can contain many inner blocks. Each inner block is once again a PL/SQL block, hence all the properties and characteristics of the inner block will be the same as outer block. The below image gives the pictorial representation of nested block structure. Parent block is the main block and child block is the nested block.

[](https://cdn.guru99.com/images/PL-SQL/110215_1221_NestedBlock1.png)

Below is the syntax for nested block.

[](https://cdn.guru99.com/images/PL-SQL/110215_1221_NestedBlock2.png)

**Syntax Explanation:**

* The above syntax shows the nested block that contains a total of two blocks.
* These blocks are labelled as 'outer\_block' and 'inner\_block'

## Scopes in Nested Block

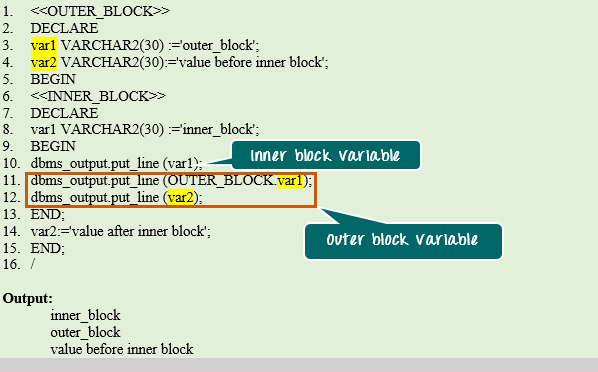
In nested block, one needs to understand the scope and visibility of each block clearly before using them. Particularly in the inner block the elements from both outer and the inner block will be visible, hence proper understanding of this is necessary.

Below points will summarize more regarding the scopes in nested blocks.

* The elements declared in the outer block and value that is defined before the inner block definition is visible inside the inner block.
* The elements declared in the inner block is not visible in the outer block. They are visible only within the inner block.
* Outer block and Inner block can have a variable with the same name.
* In case of variables with same name, inner block by default will refer to the variable declared in inner block only.
* If inner block wants to refer the outer block variable that is having the same name as that of the inner block, then outer block should be LABELLED and the outer block variable can be referred as '<outer\_block\_label>.<variable\_name>'

The below example will help to understand more about these scopes.

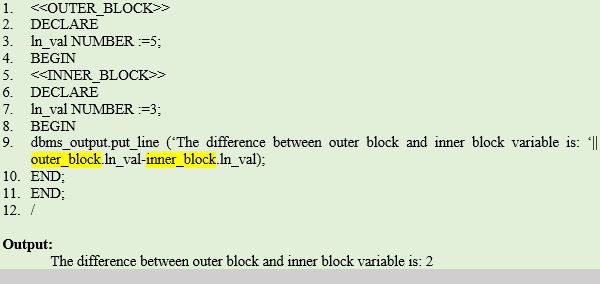
**Example 1**: In this example, we are going to see the scope of variables in the inner and outer block. Also, we are going to see how to refer the variables using block label.

[](https://cdn.guru99.com/images/PL-SQL/110215_1221_NestedBlock3.png)

**Code Explanation:**

* **Code line 1**: Labelling the outer block as "OUTER\_BLOCK".
* **Code line 3**: Declaring a variable 'var1' as VARCHAR2 (30) with the initial value of "outer block".
* **Code line 4**: Declaring a variable 'var2' as VARCHAR2 (30) with the initial value of "value before inner block".
* **Code line 6:** Labelling the inner block as "INNER\_BLOCK"
* **Code line 8:** Declaring a variable 'var1' in the inner block as VARCHAR2 (30) with the initial value of "inner block".
* **Code line 10:** Printing the value of 'var1'. Since no label is mentioned by default it will take the value from an inner block, hence printing 'inner\_block' message.
* **Code line 11:** Printing the value of outer block variable 'var1'. Since the inner block is having the variable with the same name, we need to refer with outer block label. Thus printing the message 'outer block'.
* **Code line 12:** Printing the value of outer block variable 'var2'. Since there is no variable with this name present in the inner block, by default it will take the value from an outer block, hence printing 'value before inner block' message.
* The variable 'var2' in the outer block has been assigned with the value 'value after inner block'. But this assignment has happened after the definition of an inner block, hence this value is not present in the inner block.

**Example 2**: In this example, we are going to find the difference between two numbers, one declared at the outer block and another at inner block. Both will have the same name. Let's see how block label is useful in referring these variables.

[](https://cdn.guru99.com/images/PL-SQL/110215_1221_NestedBlock4.png)

**Code Explanation:**

* **Code line 1**: Labelling the outer block as "OUTER\_BLOCK".
* **Code line 3**: Declaring a variable 'ln\_val' as NUMBER with the initial value of "5".
* **Code line 5:** Labelling the inner block as "INNER\_BLOCK"
* **Code line 7:** Declaring a variable 'ln\_val' in inner block as NUMBER with the initial value of "3".
* **Code line 9:** Printing the difference in value of 'ln\_val' from outer and inner block. The "<block\_name>.<variable\_name>" format is used to refer these variables to avoid conflicts due to same variable name.

## Variable Scope in PL/SQL:

PL/SQL allows nesting of blocks. A program block can contain another inner block. If you declare a variable within an inner block, it is not accessible to an outer block. There are two types of variable scope:

* Local Variable: Local variables are the inner block variables which are not accessible to outer blocks.
* Global Variable: Global variables are declared in outermost block.

## Example of Local and Global variables

Let's take an example to show the usage of Local and Global variables in its simple form:

1. DECLARE
2. -- Global variables
3. num1 number := 95;
4. num2 number := 85;
5. BEGIN
6. dbms\_output.put\_line('Outer Variable num1: ' || num1);
7. dbms\_output.put\_line('Outer Variable num2: ' || num2);
8. DECLARE
9. -- Local variables
10. num1 number := 195;
11. num2 number := 185;
12. BEGIN
13. dbms\_output.put\_line('Inner Variable num1: ' || num1);
14. dbms\_output.put\_line('Inner Variable num2: ' || num2);
15. END;
16. END;
17. /

After the execution, this will produce the following result:

1. Outer Variable num1: 95
2. Outer Variable num2: 85
3. Inner Variable num1: 195
4. Inner Variable num2: 185
6. PL/SQL procedure successfully completed.

PL/SQL constants

A constant is a value used in a PL/SQL block that remains unchanged throughout the program. It is a user-defined literal value. It can be declared and used instead of actual values.

Let's take an example to explain it well:

Suppose, you have to write a program which will increase the salary of the employees upto 30%, you can declare a constant and use it throughout the program. Next time if you want to increase the salary again you can change the value of constant than the actual value throughout the program.

**Syntax to declare a constant:**

1. constant\_name CONSTANT datatype := VALUE;

* **Constant\_name:**it is the name of constant just like variable name. The constant word is a reserved word and its value does not change.
* **VALUE:** it is a value which is assigned to a constant when it is declared. It can not be assigned later.

## Example of PL/SQL constant

Let's take an example to explain it well:

1. DECLARE
2. -- constant declaration
3. pi constant number := 3.141592654;
4. -- other declarations
5. radius number(5,2);
6. dia number(5,2);
7. circumference number(7, 2);
8. area number (10, 2);
9. BEGIN
10. -- processing
11. radius := 9.5;
12. dia := radius \* 2;
13. circumference := 2.0 \* pi \* radius;
14. area := pi \* radius \* radius;
15. -- output
16. dbms\_output.put\_line('Radius: ' || radius);
17. dbms\_output.put\_line('Diameter: ' || dia);
18. dbms\_output.put\_line('Circumference: ' || circumference);
19. dbms\_output.put\_line('Area: ' || area);
20. END;
21. /

After the execution of the above code at SQL prompt, it will produce the following result:.

1. Radius: 9.5
2. Diameter: 19
3. Circumference: 59.69
4. Area: 283.53
6. Pl/SQL procedure successfully completed.

# PL/SQL If

PL/SQL supports the programming language features like conditional statements and iterative statements. Its programming constructs are similar to how you use in programming languages like Java and C++.

**Syntax for IF Statement:**

There are different syntaxes for the IF-THEN-ELSE statement.

**Syntax: (IF-THEN statement):**

1. IF condition
2. THEN
3. Statement: {It is executed when condition is true}
4. END IF;

This syntax is used when you want to execute statements only when condition is TRUE.

**Syntax: (IF-THEN-ELSE statement):**

1. IF condition
2. THEN
3. {...statements to execute when condition is TRUE...}
4. ELSE
5. {...statements to execute when condition is FALSE...}
6. END IF;

This syntax is used when you want to execute one set of statements when condition is TRUE or a different set of statements when condition is FALSE.

**Syntax: (IF-THEN-ELSIF statement):**

1. IF condition1
2. THEN
3. {...statements to execute when condition1 is TRUE...}
4. ELSIF condition2
5. THEN
6. {...statements to execute when condition2 is TRUE...}
7. END IF;

This syntax is used when you want to execute one set of statements when condition1 is TRUE or a different set of statements when condition2 is TRUE.

**Syntax: (IF-THEN-ELSIF-ELSE statement):**

1. IF condition1
2. THEN
3. {...statements to execute when condition1 is TRUE...}
4. ELSIF condition2
5. THEN
6. {...statements to execute when condition2 is TRUE...}
7. ELSE
8. {...statements to execute when both condition1 and condition2 are FALSE...}
9. END IF;

It is the most advance syntax and used if you want to execute one set of statements when condition1 is TRUE, a different set of statement when condition2 is TRUE or a different set of statements when both the condition1 and condition2 are FALSE.

#### When a condition is found to be TRUE, the IF-THEN-ELSE statement will execute the corresponding code and not check the conditions any further.

#### If there no condition is met, the ELSE portion of the IF-THEN-ELSE statement will be executed.

#### ELSIF and ELSE portions are optional.

## Example of PL/SQL If Statement

Let's take an example to see the whole concept:

1. DECLARE
2. a number(3) := 500;
3. BEGIN
4. -- check the boolean condition using if statement
5. IF( a < 20 ) THEN
6. -- if condition is true then print the following
7. dbms\_output.put\_line('a is less than 20 ' );
8. ELSE
9. dbms\_output.put\_line('a is not less than 20 ' );
10. END IF;
11. dbms\_output.put\_line('value of a is : ' || a);
12. END;

After the execution of the above code in SQL prompt, you will get the following result:

a is not less than 20

value of a is : 500

PL/SQL procedure successfully completed.

# PL/SQL Case Statement

The PL/SQL CASE statement facilitates you to execute a sequence of statements based on a selector. A selector can be anything such as variable, function or an expression that the CASE statement checks to a boolean value.

The CASE statement works like the IF statement, only using the keyword WHEN. A CASE statement is evaluated from top to bottom. If it get the condition TRUE, then the corresponding THEN clause is executed and the execution goes to the END CASE clause.

**Syntax for the CASE Statement:**

1. CASE [ expression ]
2. WHEN condition\_1 THEN result\_1
3. WHEN condition\_2 THEN result\_2
4. ...
5. WHEN condition\_n THEN result\_n
6. ELSE result
7. END

## Example of PL/SQL case statement

Let's take an example to make it clear:

1. DECLARE
2. grade char(1) := 'A';
3. BEGIN
4. CASE grade
5. when 'A' then dbms\_output.put\_line('Excellent');
6. when 'B' then dbms\_output.put\_line('Very good');
7. when 'C' then dbms\_output.put\_line('Good');
8. when 'D' then dbms\_output.put\_line('Average');
9. when 'F' then dbms\_output.put\_line('Passed with Grace');
10. else dbms\_output.put\_line('Failed');
11. END CASE;
12. END;

Suppose you have stored the program in a folder you can execute it as follows:

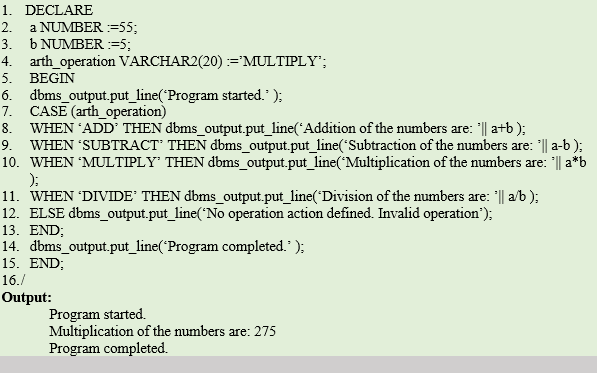
@e:/books/sql\_prgs/grade.sql;

After the execution of above code, you will get the following result:

Excellent

PL/SQL procedure successfully completed.

In this example, we are going to do arithmetic calculation between two numbers 55 and 5.

[](https://cdn.guru99.com/images/PL-SQL/110215_0814_PLSQLDecisi16.png)

**Code Explanation:**

* **Code line 2:** Declaring the variable 'a' as 'NUMBER' data type and initializing it with value '55'.
* **Code line 3:** Declaring the variable 'b' as 'NUMBER' data type and initializing it with value '5.'
* **Code line 4:** Declaring the variable 'arth\_operation' as 'VARCHAR2' data type of size 20 and initializing it with value 'MULTIPLY'.
* **Code line 6:** Printing the statement "Program started".
* **Code line 7:** CASE checks the value of the expression. In this case, the value of the variable 'arth\_operation' is 'MULTIPLY'. This value will be treated as a selector for this CASE statement now.
* **Code line 10:** The WHEN clause with value 'MULTIPLY' matches with the selector value, hence controller will select this action\_block and will print the message 'Multiplication of the numbers are: 275'.
* **Code line13:** Marks the end of CASE statement.
* **Code line14:** Printing the statement "Program completed".

**Code Output:**

Program started.

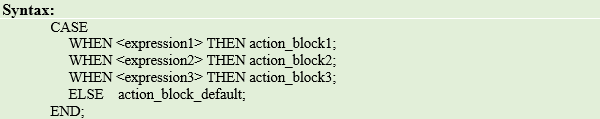
Multiplication of the numbers are: 275

Program completed.

## SEARCHED CASE Statement

SEARCHED CASE statement is similar to CASE statement, rather than using the selector to select the alternative, SEARCHED CASE will directly have the expression defined in the WHEN clause.

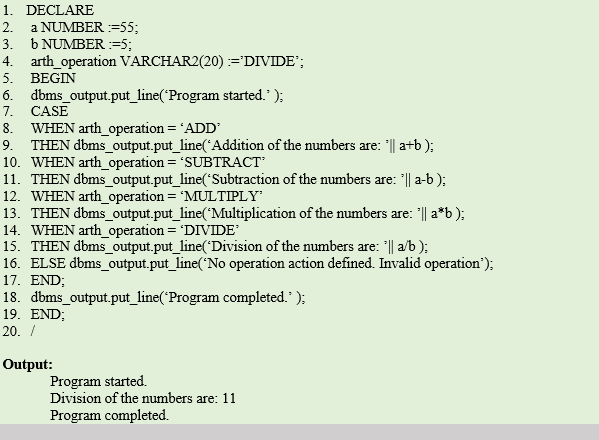
* The first WHEN clause that satisfies the condition will be executed, and the controll will skip the remaining alternatives.

[](https://cdn.guru99.com/images/PL-SQL/110215_0814_PLSQLDecisi17.png)**Syntax Explanation:**

* In the above syntax, each WHEN clause has the separate <expression> and <action\_block>.
* The WHEN clause for which the expression returns TRUE will be executed.
* 'ELSE' block is optional which hold the <action\_block\_default> that needs to be executed when none of the alternatives satisfies.
* The 'END' marks the end of CASE statement and it is a mandatory part of CASE.

**Example 1:** Arithmatic Calculation using Searched Case

In this example, we are going to do arithmetic calculation between two numbers 55 and 5.

[](https://cdn.guru99.com/images/PL-SQL/110215_0814_PLSQLDecisi18.png)

**Code Explanation:**

* **Code line 2:** Declaring the variable 'a' as 'NUMBER' data type and initializing it with value '55'.
* **Code line 3:** Declaring the variable 'b' as 'NUMBER' data type and initializing it with value '5'.
* **Code line 4:** Declaring the variable 'arth\_operation' as 'VARCHAR2' data type of size 20 and initializing it with value 'DIVIDE.'
* **Code line 6:** Printing the statement "Program started".
* **Code line 7:** SEARCHED CASE statement begins.The code from line8 to line 13 is skipped as their selector value (ADD, SUBTRACT, MULTIPLY) doesn't match with the value of 'arth\_operation'.
* **Code line 14:** The WHEN clause expression "arth\_operation = 'DIVIDE'" satisfied and the expression returns TRUE.
* **Code line 15:** Action\_block of the WHEN clause will be executed, and message 'Division of the numbers are: 11' will be printed.
* **Code line 17:** Marks the end of CASE statement.
* **Code line 18:** Printing the statement "Program completed".

**Code Output:**

Program started.

Division of the numbers are: 11

Program completed.

# PL/SQL Loop

The PL/SQL loops are used to repeat the execution of one or more statements for specified number of times. These are also known as iterative control statements.

**Syntax for a basic loop:**

1. LOOP
2. Sequence of statements;
3. END LOOP;

## Types of PL/SQL Loops

There are 4 types of PL/SQL Loops.

1. Basic Loop / Exit Loop
2. While Loop
3. For Loop
4. Cursor For Loop

# PL/SQL Exit Loop (Basic Loop)

PL/SQL exit loop is used when a set of statements is to be executed at least once before the termination of the loop. There must be an EXIT condition specified in the loop, otherwise the loop will get into an infinite number of iterations. After the occurrence of EXIT condition, the process exits the loop.

**Syntax of basic loop:**

1. LOOP
2. Sequence of statements;
3. END LOOP;

**Syntax of exit loop:**

1. LOOP
2. statements;
3. EXIT;
4. {or EXIT WHEN condition;}
5. END LOOP;

## Example of PL/SQL EXIT Loop

Let's take a simple example to explain it well:

1. DECLARE
2. i NUMBER := 1;
3. BEGIN
4. LOOP
5. EXIT WHEN i>10;
6. DBMS\_OUTPUT.PUT\_LINE(i);
7. i := i+1;
8. END LOOP;
9. END;

After the execution of the above code, you will get the following result:

1

2

3

4

5

6

7

8

9

10

Note: You must follow these steps while using PL/SQL Exit Loop.

* Initialize a variable before the loop body
* Increment the variable in the loop.
* You should use EXIT WHEN statement to exit from the Loop. Otherwise the EXIT statement without WHEN condition, the statements in the Loop is executed only once.

## PL/SQL EXIT Loop Example 2

1. DECLARE
2. VAR1 NUMBER;
3. VAR2 NUMBER;
4. BEGIN
5. VAR1:=100;
6. VAR2:=1;
7. LOOP
8. DBMS\_OUTPUT.PUT\_LINE (VAR1\*VAR2);
9. IF (VAR2=10) THEN
10. EXIT;
11. END IF;
12. VAR2:=VAR2+1;
13. END LOOP;
14. END;

Output:

100

200

300

400

500

600

700

800

900

1000

# PL/SQL While Loop

PL/SQL while loop is used when a set of statements has to be executed as long as a condition is true, the While loop is used. The condition is decided at the beginning of each iteration and continues until the condition becomes false.

**Syntax of while loop:**

1. WHILE <condition>
2. LOOP statements;
3. END LOOP;

## Example of PL/SQL While Loop

Let's see a simple example of PL/SQL WHILE loop.

1. DECLARE
2. i INTEGER := 1;
3. BEGIN
4. WHILE i <= 10 LOOP
5. DBMS\_OUTPUT.PUT\_LINE(i);
6. i := i+1;
7. END LOOP;
8. END;

After the execution of the above code, you will get the following result:

1

2

3

4

5

6

7

8

9

10

Note: You must follow these steps while using PL/SQL WHILE Loop.

* Initialize a variable before the loop body.
* Increment the variable in the loop.
* You can use EXIT WHEN statements and EXIT statements in While loop but it is not done often.

## PL/SQL WHILE Loop Example 2

1. DECLARE
2. VAR1 NUMBER;
3. VAR2 NUMBER;
4. BEGIN
5. VAR1:=200;
6. VAR2:=1;
7. WHILE (VAR2<=10)
8. LOOP
9. DBMS\_OUTPUT.PUT\_LINE (VAR1\*VAR2);
10. VAR2:=VAR2+1;
11. END LOOP;
12. END;

Output:

200

400

600

800

1000

1200

1400

1600

1800

2000

# PL/SQL FOR Loop

PL/SQL for loop is used when when you want to execute a set of statements for a predetermined number of times. The loop is iterated between the start and end integer values. The counter is always incremented by 1 and once the counter reaches the value of end integer, the loop ends.

**Syntax of for loop:**

1. FOR counter IN initial\_value .. final\_value LOOP
2. LOOP statements;
3. END LOOP;

* initial\_value : Start integer value
* final\_value : End integer value

## PL/SQL For Loop Example 1

Let's see a simple example of PL/SQL FOR loop.

1. BEGIN
2. FOR k IN 1..10 LOOP
3. -- note that k was not declared
4. DBMS\_OUTPUT.PUT\_LINE(k);
5. END LOOP;
6. END;

After the execution of the above code, you will get the following result:

1

2

3

4

5

6

7

8

9

10

Note: You must follow these steps while using PL/SQL WHILE Loop.

* You don't need to declare the counter variable explicitly because it is declared implicitly in the declaration section.
* The counter variable is incremented by 1 and does not need to be incremented explicitly.
* You can use EXIT WHEN statements and EXIT statements in FOR Loops but it is not done often.

## PL/SQL For Loop Example 2

1. DECLARE
2. VAR1 NUMBER;
3. BEGIN
4. VAR1:=10;
5. FOR VAR2 IN 1..10
6. LOOP
7. DBMS\_OUTPUT.PUT\_LINE (VAR1\*VAR2);
8. END LOOP;
9. END;

Output:

10

20

30

40

50

60

70

80

90

100

## PL/SQL For Loop REVERSE Example 3

Let's see an example of PL/SQL for loop where we are using REVERSE keyword.

1. DECLARE
2. VAR1 NUMBER;
3. BEGIN
4. VAR1:=10;
5. FOR VAR2 IN REVERSE 1..10
6. LOOP
7. DBMS\_OUTPUT.PUT\_LINE (VAR1\*VAR2);
8. END LOOP;
9. END;

Output:

100

90

80

70

60

50

40

30

20

10

# PL/SQL Continue Statement

The continue statement is used to exit the loop from the reminder if its body either conditionally or unconditionally and forces the next iteration of the loop to take place, skipping any codes in between.

The continue statement is not a keyword in Oracle 10g. It is a new feature encorporated in oracle 11g.

For example: If a continue statement exits a cursor FOR LOOP prematurely then it exits an inner loop and transfer control to the next iteration of an outer loop, the cursor closes (in this context, CONTINUE works like GOTO).

**Syntax:**

1. continue;

## Example of PL/SQL continue statement

Let's take an example of PL/SQL continue statement.

1. DECLARE
2. x NUMBER := 0;
3. BEGIN
4. LOOP -- After CONTINUE statement, control resumes here
5. DBMS\_OUTPUT.PUT\_LINE ('Inside loop:  x = ' || TO\_CHAR(x));
6. x := x + 1;
7. IF x < 3 THEN
8. CONTINUE;
9. END IF;
10. DBMS\_OUTPUT.PUT\_LINE
11. ('Inside loop, after CONTINUE:  x = ' || TO\_CHAR(x));
12. EXIT WHEN x = 5;
13. END LOOP;
15. DBMS\_OUTPUT.PUT\_LINE (' After loop:  x = ' || TO\_CHAR(x));
16. END;
17. /

After the execution of above code, you will get the following result:

Inside loop: x = 0

Inside loop: x = 1

Inside loop: x = 2

Inside loop, after CONTINUE: x = 3

Inside loop: x = 3

Inside loop, after CONTINUE: x = 4

Inside loop: x = 4

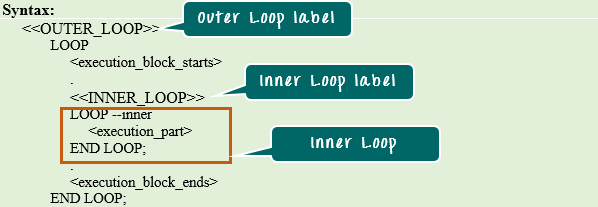
Inside loop, after CONTINUE: x = 5

After loop: x = 5

#### Note: The continue statement is not supported in Oracle 10g. Oracle 11g supports this as a new feature.

## Labelling of Loops

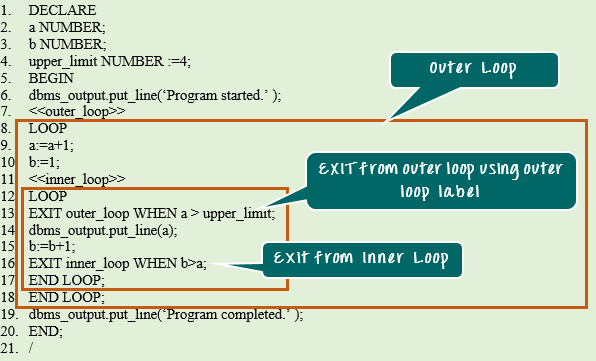
In PL/SQL, the loops can be labeled. The label should be enclosed between "<<" and ">>". The labeling of loops particularly in nested loop codes will give more readability. The label can be given in EXIT command to exit from that particular loop. Using label, the control can be made to directly exit the outer loop of the nested loops from anyplace inside the loops, by giving the exit command followed by outer loop label.

[](https://cdn.guru99.com/images/PL-SQL/110215_0850_LoopsConcep11.png)

**Syntax Explanation:**

* In the above syntax, the out loop has one more loop inside it.
* The '<<OUTER\_LOOP>>' and '<<INNER\_LOOP>>' are the labels of these loops.

**Example 1**: In this example, we are going to print number starting from 1 using Basic loop statement. Each number will be printed as many times as its value. The upper limit of the series is fixed at the program declaration part. Let us learn how we can use the label concept to achieve this. For that, we will execute the following code

[](https://cdn.guru99.com/images/PL-SQL/110215_0850_LoopsConcep12.png)[](https://cdn.guru99.com/images/PL-SQL/110215_0850_LoopsConcep13.png)

**Code Explanation:**

* **Code line 2-3**: Declaring the variable 'a' and 'b' as 'NUMBER' data type.
* **Code line 4**: Declaring the variable 'upper\_limit' as 'NUMBER' data type with value '4'
* **Code line 6**: Printing the statement "Program started".
* **Code line 7:** The outer loop has been labeled as "outer\_loop"
* **Code line 9:** The value of 'a' is incremented by 1.
* **Code line 11:** Inner loop has been labeled as "inner\_loop".
* **Code line 13:** EXIT condition that check whether the value 'a' is higher than 'upper\_limit' value. If not then it will go further, else it exits outer loop directly.
* **Code line 14:** Printing the value of 'b'.
* **Code line 15:** Increments the value of 'b' by +1.
* **Code line 16:** EXIT condition that checks whether the value of 'b' is higher than 'a'. If so, then it will exit the control from the inner loop.
* **Code line 14:** Printing the statement "Program completed"

|  |  |  |
| --- | --- | --- |
| **Loop** | **EXIT Criteria** | **Usage** |
| Basic Loop | Exit when encounters the keyword 'EXIT' in the execution part | Good to use when exit is not based on any particular condition. |
| WHILE Loop | Exit when the check condition returns false | Good to use when the loop count is unknown, and exit is based on some other condition. |
| FOR Loop | Exit when the counter reaches the limit | Good to use when loop count to be executed is known. |

# PL/SQL GOTO Statement

In PL/SQL, GOTO statement makes you able to get an unconditional jump from the GOTO to a specific executable statement label in the same subprogram of the PL/SQL block.

Here the label declaration which contains the label\_name encapsulated within the <<>> symbol and must be followed by at least one statement to execute.

**Syntax:**

1. GOTO label\_name;

Here the label declaration which contains the label\_name encapsulated within the <<>> symbol and must be followed by at least one statement to execute.

1. GOTO label\_name;
2. ..
3. ..
4. <<label\_name>>
5. Statement;

## Example of PL/SQL GOTO statement

Let's take an example of PL/SQL GOTO statement.

1. DECLARE
2. a number(2) := 30;
3. BEGIN
4. <<loopstart>>
5. -- while loop execution
6. WHILE a < 50 LOOP
7. dbms\_output.put\_line ('value of a: ' || a);
8. a := a + 1;
9. IF a = 35 THEN
10. a := a + 1;
11. GOTO loopstart;
12. END IF;
13. END LOOP;
14. END;
15. /

After the execution of above code, you will get the following result:

value of a: 30

value of a: 31

value of a: 32

value of a: 33

value of a: 34

value of a: 36

value of a: 37

value of a: 38

value of a: 39

value of a: 40

value of a: 41

value of a: 42

value of a: 43

value of a: 44

value of a: 45

value of a: 46

value of a: 47

value of a: 48

value of a: 49

Statement processed.

## Restriction on GOTO statement

Following is a list of some restrictions imposed on GOTO statement.

* Cannot transfer control into an IF statement, CASE statement, LOOP statement or sub-block.
* Cannot transfer control from one IF statement clause to another or from one CASE statement WHEN clause to another.
* Cannot transfer control from an outer block into a sub-block.
* Cannot transfer control out of a subprogram.
* Cannot transfer control into an exception handler.

# PL/SQL Procedure

The PL/SQL stored procedure or simply a procedure is a PL/SQL block which performs one or more specific tasks. It is just like procedures in other programming languages.

The procedure contains a header and a body.

* **Header:** The header contains the name of the procedure and the parameters or variables passed to the procedure.
* **Body:** The body contains a declaration section, execution section and exception section similar to a general PL/SQL block.

## How to pass parameters in procedure:

When you want to create a procedure or function, you have to define parameters .There is three ways to pass parameters in procedure:

1. **IN parameters:** The IN parameter can be referenced by the procedure or function. The value of the parameter cannot be overwritten by the procedure or the function.
2. **OUT parameters:** The OUT parameter cannot be referenced by the procedure or function, but the value of the parameter can be overwritten by the procedure or function.
3. **INOUT parameters:** The INOUT parameter can be referenced by the procedure or function and the value of the parameter can be overwritten by the procedure or function.

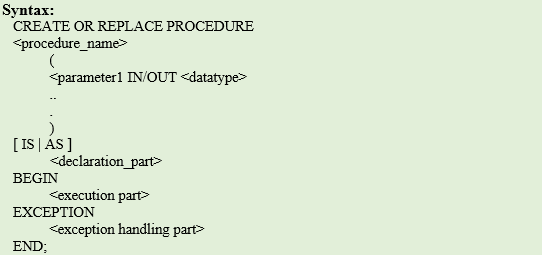
#### A procedure may or may not return any value.

## Procedure

Procedure is a subprogram unit that consists of a group of PL/SQL statements. Each procedure in Oracle has its own unique name by which it can be referred. This subprogram unit is stored as a database object. Below are the characteristics of this subprogram unit.

**Note:** Subprogram is nothing but a procedure, and it needs to be created manually as per the requirement. Once created they will be stored as database objects.

* Procedures are standalone blocks of a program that can be stored in the database.
* Call to these procedures can be made by referring to their name, to execute the PL/SQL statements.
* It is mainly used to execute a process in PL/SQL.
* It can have nested blocks, or it can be defined and nested inside the other blocks or packages.
* It contains declaration part (optional), execution part, exception handling part (optional).
* The values can be passed into the procedure or fetched from the procedure through parameters.
* These parameters should be included in the calling statement.
* Procedure can have a RETURN statement to return the control to the calling block, but it cannot return any values through the RETURN statement.
* Procedures cannot be called directly from SELECT statements, they can be called from another block or through EXEC keyword.

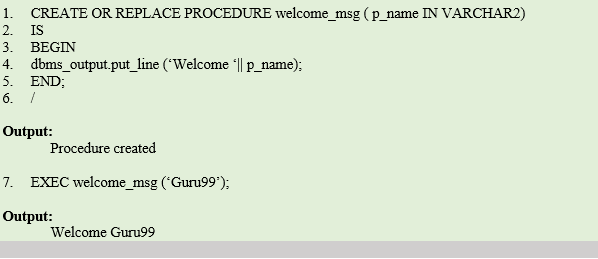
[](https://cdn.guru99.com/images/PL-SQL/110215_0958_Subprograms1.png)

**Syntax Explanation:**

* CREATE PROCEDURE instructs the compiler to create new procedure. Keyword 'OR REPLACE' instructs the compile to replace the existing procedure (if any) with the current one.
* Procedure name should be unique.
* Keyword 'IS' will be used, when the procedure is nested into some other blocks. If the procedure is standalone then 'AS' will be used. Other than this coding standard, both have the same meaning.

**Example1: Creating Procedure and calling it using EXEC**

In this example, we are going to create a procedure that takes the name as input and prints the welcome message as output. We are going to use EXEC command to call procedure.

[](https://cdn.guru99.com/images/PL-SQL/110215_0958_Subprograms2.png)

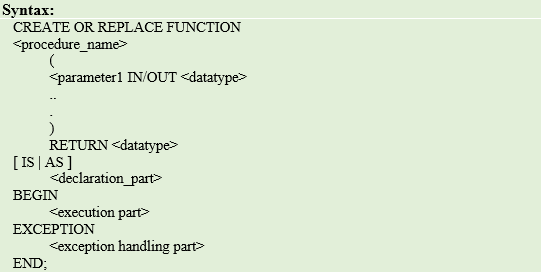
**Code Explanation:**

* **Code line 1**: Creating the procedure with name 'welcome\_msg' and with one parameter 'p\_name' of 'IN' type.
* **Code line 4**: Printing the welcome message by concatenating the input name.
* Procedure is compiled successfully.
* **Code line 7**: Calling the procedure using EXEC command with the parameter 'Guru99'. Procedure is executed, and the message is printed out as "Welcome Guru99".

## Function

Functions is a standalone PL/SQL subprogram. Like PL/SQL procedure, functions has a unique name by which it can be referred. These are stored as PL/SQL database objects. Below are some of the characteristics of functions.

* Functions are a standalone block that is mainly used for calculation purpose.
* Function use RETURN keyword to return the value, and the datatype of this is defined at the time of creation.
* Function should either return a value or raise the exception, i.e. return is mandatory in functions.
* Function with no DML statements can be directly called in SELECT query whereas the function with DML operation can only be called from other PL/SQL blocks.
* It can have nested blocks, or it can be defined and nested inside the other blocks or packages.
* It contains declaration part (optional), execution part, exception handling part (optional).
* The values can be passed into the function or fetched from the procedure through the parameters.
* These parameters should be included in the calling statement.
* Function can also return the value through OUT parameters other than using RETURN.
* Since it will always return the value, in calling statement it always accompany with assignment operator to populate the variables.

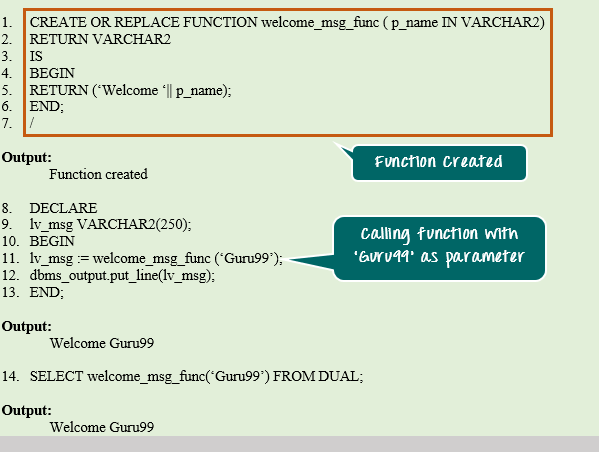
[](https://cdn.guru99.com/images/PL-SQL/110215_0958_Subprograms3.png)

**Syntax Explanation:**

* CREATE FUNCTION instructs the compiler to create a new function. Keyword 'OR REPLACE' instructs the compiler to replace the existing function (if any) with the current one.
* Function name should be unique.
* RETURN datatype should be mentioned.
* Keyword 'IS' will be used, when the procedure is nested into some other blocks. If the procedure is standalone then 'AS' will be used. Other than this coding standard, both have the same meaning.

**Example1: Creating Function and calling it using Anonymous Block**

In this program, we are going to create a function that takes the name as input and returns the welcome message as output. We are going to use anonymous block and select statement to call the function.

[](https://cdn.guru99.com/images/PL-SQL/110215_0958_Subprograms4.png)

**Code Explanation:**

* **Code line 1**: Creating the function with name 'welcome\_msg\_func' and with one parameter 'p\_name' of 'IN' type.
* **Code line 2**: declaring the return type as VARCHAR2
* **Code line 5**: Returning the concatenated value 'Welcome' and the parameter value.
* **Code line 8**: Anonymous block to call the above function.
* **Code line 9**: Declaring the variable with datatype same as the return datatype of the function.
* **Code line 11**: Calling the function and populating the return value to the variable 'lv\_msg'.
* **Code line 12**: Printing the variable value. The output you will get here is "Welcome Guru99"
* **Code line 14**: Calling the same function through SELECT statement. The return value is directed to the standard output directly.

## Similarities between Procedure and Function

* Both can be called from other PL/SQL blocks.
* If the exception raised in the subprogram is not handled in the subprogram exception handling section, then it will propagate to the calling block.
* Both can have as many parameters as required.
* Both are treated as database objects in PL/SQL.

## Difference between Procedure and Function

|  |  |
| --- | --- |
| **Procedure** | **Function** |
| * Used mainly to execute certain process | * Used mainly to perform some calculation |
| * Cannot called in SELECT statement | * Function that contain no DML statements can be called in SELECT statement |
| * Use OUT parameter to return the value | * Use RETURN to return the value |
| * It is not mandatory to return the value | * It is mandatory to return the value |
| * RETURN will simply exit the control from subprogram. | * RETURN will exit the control from subprogram and also returns the value |
| * Return datatype will not be specified at the time of creation | * Return datatype is mandatory at the time of creation |

## Built-in Functions in PL/SQL

PL/SQL contains various built-in functions to work with strings and date datatype. Here we are going to see the commonly used functions and their usage

### Conversion Functions

These built-in functions are used to convert one datatype to another datatype.

## PL/SQL Create Procedure

**Syntax for creating procedure:**

1. CREATE [OR REPLACE] PROCEDURE procedure\_name
2. [ (parameter [,parameter]) ]
3. IS
4. [declaration\_section]
5. BEGIN
6. executable\_section
7. [EXCEPTION
8. exception\_section]
9. END [procedure\_name];

## Create procedure example

In this example, we are going to insert record in user table. So you need to create user table first.

**Table creation:**

1. create table user(id number(10) primary key,name varchar2(100));

Now write the procedure code to insert record in user table.

**Procedure Code:**

1. create or replace procedure "INSERTUSER"
2. (id IN NUMBER,
3. name IN VARCHAR2)
4. is
5. begin
6. insert into user values(id,name);
7. end;
8. /

Output:

Procedure created.

## PL/SQL program to call procedure

Let's see the code to call above created procedure.

1. BEGIN
2. insertuser(101,'Rahul');
3. dbms\_output.put\_line('record inserted successfully');
4. END;
5. /

Now, see the "USER" table, you will see one record is inserted.

|  |  |
| --- | --- |
| **ID** | **Name** |
| 101 | Rahul |

## PL/SQL Drop Procedure

**Syntax for drop procedure**

1. DROP PROCEDURE procedure\_name;

## Example of drop procedure

1. DROP PROCEDURE pro1;

# PL/SQL Function

The PL/SQL Function is very similar to PL/SQL Procedure. The main difference between procedure and a function is, a function must always return a value, and on the other hand a procedure may or may not return a value. Except this, all the other things of PL/SQL procedure are true for PL/SQL function too.

**Syntax to create a function:**

1. CREATE [OR REPLACE] FUNCTION function\_name [parameters]
2. [(parameter\_name [IN | OUT | IN OUT] type [, ...])]
3. RETURN return\_datatype
4. {IS | AS}
5. BEGIN
6. < function\_body >
7. END [function\_name];

**Here:**

* **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.
* The AS keyword is used instead of the IS keyword for creating a standalone function.

## PL/SQL Function Example

Let's see a simple example to **create a function**.

1. create or replace function adder(n1 in number, n2 in number)
2. return number
3. is
4. n3 number(8);
5. begin
6. n3 :=n1+n2;
7. return n3;
8. end;
9. /

Now write another program to **call the function**.

1. DECLARE
2. n3 number(2);
3. BEGIN
4. n3 := adder(11,22);
5. dbms\_output.put\_line('Addition is: ' || n3);
6. END;
7. /

**Output:**

Addition is: 33

Statement processed.

0.05 seconds

## Another PL/SQL Function Example

Let's take an example to demonstrate Declaring, Defining and Invoking a simple PL/SQL function which will compute and return the maximum of two values.

1. DECLARE
2. a number;
3. b number;
4. c number;
5. FUNCTION findMax(x IN number, y IN number)
6. RETURN number
7. IS
8. z number;
9. BEGIN
10. IF x > y THEN
11. z:= x;
12. ELSE
13. Z:= y;
14. END IF;
16. RETURN z;
17. END;
18. BEGIN
19. a:= 23;
20. b:= 45;
22. c := findMax(a, b);
23. dbms\_output.put\_line(' Maximum of (23,45): ' || c);
24. END;
25. /

**Output:**

Maximum of (23,45): 45

Statement processed.

0.02 seconds

## PL/SQL function example using table

Let's take a customer table. This example illustrates creating and calling a standalone function. This function will return the total number of CUSTOMERS in the customers table.

#### Create customers table and have records in it.

|  |  |  |  |
| --- | --- | --- | --- |
| **Customers** | | | |
| **Id** | **Name** | **Department** | **Salary** |
| 1 | alex | web developer | 35000 |
| 2 | ricky | program developer | 45000 |
| 3 | mohan | web designer | 35000 |
| 4 | dilshad | database manager | 44000 |

**Create Function:**

1. CREATE OR REPLACE FUNCTION totalCustomers
2. RETURN number IS
3. total number(2) := 0;
4. BEGIN
5. SELECT count(\*) into total
6. FROM customers;
7. RETURN total;
8. END;
9. /

After the execution of above code, you will get the following result.

Function created.

**Calling PL/SQL Function:**

While creating a function, you have to give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task. Once the function is called, the program control is transferred to the called function.

After the successful completion of the defined task, the call function returns program control back to the main program.

To call a function you have to pass the required parameters along with function name and if function returns a value then you can store returned value. Following program calls the function totalCustomers from an anonymous block:

1. DECLARE
2. c number(2);
3. BEGIN
4. c := totalCustomers();
5. dbms\_output.put\_line('Total no. of Customers: ' || c);
6. END;
7. /

After the execution of above code in SQL prompt, you will get the following result.

Total no. of Customers: 4

PL/SQL procedure successfully completed.

## PL/SQL Recursive Function

You already know that a program or a subprogram can call another subprogram. When a subprogram calls itself, it is called recursive call and the process is known as recursion.

## Example to calculate the factorial of a number

Let's take an example to calculate the factorial of a number. This example calculates the factorial of a given number by calling itself recursively.

1. DECLARE
2. num number;
3. factorial number;
5. FUNCTION fact(x number)
6. RETURN number
7. IS
8. f number;
9. BEGIN
10. IF x=0 THEN
11. f := 1;
12. ELSE
13. f := x \* fact(x-1);
14. END IF;
15. RETURN f;
16. END;
18. BEGIN
19. num:= 6;
20. factorial := fact(num);
21. dbms\_output.put\_line(' Factorial '|| num || ' is ' || factorial);
22. END;
23. /

After the execution of above code at SQL prompt, it produces the following result.

Factorial 6 is 720

PL/SQL procedure successfully completed.

## PL/SQL Drop Function

**Syntax for removing your created function:**

If you want to remove your created function from the database, you should use the following syntax.

1. DROP FUNCTION function\_name;

## Introducing to PL/SQL function

PL/SQL function is a named block that returns a value. A PL/SQL function is also known as a subroutine or a subprogram. To create a PL/SQL function, you use the following syntax:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | CREATE[ORREPLACE]FUNCTIONfunction\_name[(     parameter\_1[IN][OUT]data\_type,     parameter\_2[IN][OUT]data\_type,     parameter\_N[IN][OUT]data\_type]      RETURNreturn\_data\_typeIS  --the declaration statements  BEGIN     -- the executable statements     returnreturn\_data\_type;     EXCEPTION      -- the exception-handling statements  END;  / |

Let’s examine the syntax of creating a function in greater detail:

You specify the function name function\_name after the FUNCTION keyword. By convention, the function name should start with a verb, for example convert\_to\_number .

A function may have zero or more than one parameter. You specify the parameter names in the parameter\_1, parameter\_2, etc. You must specify the data type of each parameter explicitly in the data\_type. Each parameter has one of three modes: IN, OUT and IN OUT.

* An IN parameter is a read-only parameter. If the function tries to change the value of the IN parameters, the compiler will issue an error message. You can pass a constant, literal, initialized variable, or expression to the function as the IN parameter.
* An OUT parameter is a write-only parameter. The OUT parameters are used to return values back to the calling program. An OUT parameter is initialized to a default value of its type when the function begins regardless of its original value before being passed to the function.
* An IN OUT parameter is read and write parameter. It means the function reads the value from an IN OUT parameter, change its value and return it back to the calling program.

The function must have at least one RETURN statement in the execution section. The RETURN clause in the function header specifies the data type of returned value.

The [block structure](http://www.plsqltutorial.com/plsql-block-structure/) of a function is similar to an anonymous block with an additional function header section.

## Examples of PL/SQL Function

We are going to create a function named try\_parse that parses a string and returns a number if the input string is a number or NULL if it cannot be converted to a number.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | CREATEORREPLACEFUNCTIONtry\_parse(      iv\_numberINVARCHAR2)    RETURNNUMBERIS  BEGIN     RETURNto\_number(iv\_number);     EXCEPTION       WHENothersTHEN          RETURNNULL;  END; |

The iv\_number is an IN parameter whose data type is VARCHAR2 so that you can pass any string to the   try\_parse() function.

Inside the function, we used the built-in PL/SQL function named  to\_number() to convert a string into a number. If any exception occurs, the function returns NULL in the exception section, otherwise, it returns a number.

## Calling PL/SQL Function

The PL/SQL function returns a value so you can use it on the right-hand side of an assignment or in a SELECT statement.

Let’s create an anonymous block to use the try\_parse() function.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | SETSERVEROUTPUTONSIZE1000000;  DECLARE    n\_xnumber;    n\_ynumber;     n\_znumber;  BEGIN     n\_x:=try\_parse('574');     n\_y:=try\_parse('12.21');     n\_z:=try\_parse('abcd');       DBMS\_OUTPUT.PUT\_LINE(n\_x);     DBMS\_OUTPUT.PUT\_LINE(n\_y);     DBMS\_OUTPUT.PUT\_LINE(n\_z);  END;  / |

We can also use the try\_parse() function in a SELECT statement as follows:

|  |  |
| --- | --- |
| 1  2  3 | SELECTtry\_parse('1234')FROMdual;    SELECTtry\_parse('Abc')FROMdual; |

Notice that dual table a special one-row table that is used for selecting pseudo-column like our examples above.

SELECT statement in PL/SQL

The select statement retrieves data from Oracle table and place the data in temporary variables:

Syntax:

SELECT columnnames..

INTO variablenames/Record name

FROM tablename

WHERE conditions;

The columnnames must contain at least one column and may include arithmetic or string expressions, group functions etc. The variablenames must contain a list of variables to hold the values returned by the SELECT clause. The Recordname is a PL/SQL record type.

If no rows are returned, the standard exception (error condition) NO\_DATA\_FOUND occurs.

If more than one rows are retrieved, the TOO\_MANY\_ROWS exception occurs.

|  |
| --- |
| declare  n\_eid employee.eid%type;  v\_empname employee.empname%type;  v\_street employee.street%type;  v\_city employee.city%type;  begin  select eid,empname,street,city  into n\_eid,v\_empname,v\_street,v\_city  from employee  where eid=109;  dbms\_output.put\_line('emp id '||n\_eid);  dbms\_output.put\_line('emp name '||v\_empname);  dbms\_output.put\_line('emp street '||v\_street);  dbms\_output.put\_line('city '||v\_city);  end;  / |

In the above program the first line is declare statement. In the 2,3,4 and 5th line the temporary variables are declared and are declared to hold data from table employee.

In the execution part the select statement is used to extract the data of employee id=109 and place the data in the temporary variables.

Finally the data obtained is displayed. The output is:

|  |
| --- |
| SQL> @e:/books/sql\_prgs/selectinto.sql  emp id 109  emp name ananthi  emp street rajaji street  city chennai  PL/SQL procedure successfully completed. |

In this since the row returned is only a single row no problem arises. But if the select coloum returns more than one row then error occurs unless it is handled properly.

## DML Transactions in PL/SQL

DML stands for **Data Manipulation Language**. These statements are mainly used to perform the manipulation activity. It basically deals with the below operations.

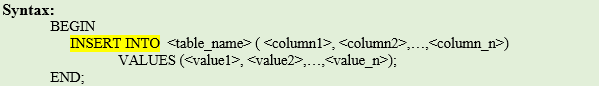
* Data Insertion
* Data Update
* Data Deletion
* Data Projection/Fetching

In PL/SQL, we can do the data manipulation only by using the [SQL](https://www.guru99.com/sql.html) commands.

### Data Insertion

In PL/SQL, we can insert the data into any table using the [SQL](https://www.guru99.com/sql.html) command INSERT INTO. This command will take the table name, table column and column values as the input and insert the value in the base table.

The INSERT command can also take the values directly from another table using 'SELECT' statement rather than giving the values for each column. Through 'SELECT' statement, we can insert as many rows as the base table contains.

[](https://cdn.guru99.com/images/PL-SQL/110215_1042_SQLinPLSQL1.png)

**Syntax Explanation:**

* The above syntax shows the INSERT INTO command. The table name and values are mandatory field, whereas column names are not mandatory if the insert statements have values for all the column of the table.
* In the following program INSERT command is used within PL/SQL to insert a row into the table

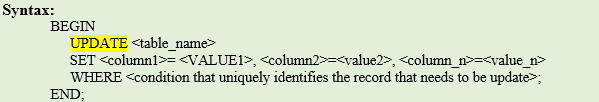
|  |
| --- |
| declare  n\_eid employee.eid%type:=&n\_eid;  v\_empname employee.empname%type:='&v\_empname';  v\_street employee.street%type:='&v\_street';  v\_city employee.city%type:='&v\_city';  begin  insert into employee (eid,empname,street,city) values(n\_eid,v\_empname,v\_street,v\_city);  commit;  end;  / |

* Output:

|  |
| --- |
| SQL> @e:/books/sql\_prgs/insertplsql.sql  Enter value for n\_eid: 110  old 2: n\_eid employee.eid%type:=&n\_eid;  new 2: n\_eid employee.eid%type:=110;  Enter value for v\_empname: kailash  old 3: v\_empname employee.empname%type:='&v\_empname';  new 3: v\_empname employee.empname%type:='kailash';  Enter value for v\_street: 3rd street  old 4: v\_street employee.street%type:='&v\_street';  new 4: v\_street employee.street%type:='3rd street';  Enter value for v\_city: chennai  old 5: v\_city employee.city%type:='&v\_city';  new 5: v\_city employee.city%type:='chennai';  PL/SQL procedure successfully completed. |

### Data Update

Data update simply means an update of the value of any column in the table. This can be done using 'UPDATE' statement. This statement takes the table name, column name and value as the input and updates the data.

[](https://cdn.guru99.com/images/PL-SQL/110215_1042_SQLinPLSQL3.png)

**Syntax Explanation:**

* The above syntax shows the UPDATE. The keyword 'SET' instruct that PL/SQL engine to update the value of the column with the value given.
* 'WHERE' clause is optional. If this clause is not given, then the value of the mentioned column in the entire table will be updated.

UPDATE commend:

* The UPDATE is used in a PL/SQL block for modification of data.

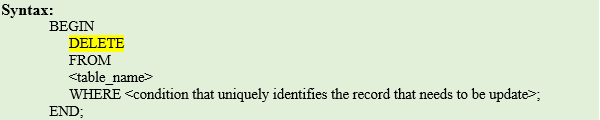
|  |
| --- |
| declare  n\_eid employee.eid%type:=&n\_eid;  begin  update employee  set city='calcutta'  where eid=n\_eid;  commit;  end;  / |

Output:

|  |
| --- |
| SQL> @e:/books/sql\_prgs/updateplsql.sql  Enter value for n\_eid: 100  old 2: n\_eid employee.eid%type:=&n\_eid;  new 2: n\_eid employee.eid%type:=100;  PL/SQL procedure successfully completed. |

### Data Deletion

Data deletion means to delete one full record from the database table. The 'DELETE' command is used for this purpose.

[](https://cdn.guru99.com/images/PL-SQL/110215_1042_SQLinPLSQL4.png)

**Syntax Explanation:**

* The above syntax shows the DELETE command. The keyword 'FROM' is optional and with or without 'FROM' clause the command behaves in the same way.
* 'WHERE' clause is optional. If this clause is not given, then the entire table will be deleted.

DELETE is used in PL/SQL block to remove rows.

|  |
| --- |
| declare  n\_eid employee.eid%type:=&n\_eid;  begin  delete from employee where eid=n\_eid;  commit;  end;  / |

* Output:

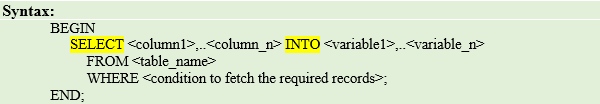
|  |
| --- |
| SQL> @e:/books/sql\_prgs/deleteplsql.sql  Enter value for n\_eid: 110  old 2: n\_eid employee.eid%type:=&n\_eid;  new 2: n\_eid employee.eid%type:=110;  PL/SQL procedure successfully completed. |

### Data Projection/Fetching

Data projection/fetching means to retrieve the required data from the database table. This can be achieved by using the command 'SELECT' with 'INTO' clause. The 'SELECT' command will fetch the values from the database, and 'INTO' clause will assign these values to the local variable of the PL/SQL block.

Below are the points that need to be considered in 'SELECT' statement.

* 'SELECT' statement should return only one record while using 'INTO' clause as one variable can hold only one value. If the 'SELECT' statement returns more than one value than 'TOO\_MANY\_ROWS' exception will be raised.
* 'SELECT' statement will assign the value to the variable in the 'INTO' clause, so it needs to get at least one record from the table to populate the value. If it didn't get any record, then the exception 'NO\_DATA\_FOUND' is raised.
* The number of columns and their datatype in 'SELECT' clause should match with the number of variables and their datatypes in the 'INTO' clause.
* The values are fetched and populated in the same order as mentioned in the statement.
* 'WHERE' clause is optional that allows to have more restriction on the records that is going to be fetched.
* 'SELECT' statement can be used in the 'WHERE' condition of other DML statements to define the values of the conditions.
* The 'SELECT' statement when using 'INSERT', 'UPDATE', 'DELETE' statements should not have 'INTO' clause as it will not populate any variable in these cases.

[](https://cdn.guru99.com/images/PL-SQL/110215_1042_SQLinPLSQL5.png)

**Syntax Explanation:**

* The above syntax shows the SELECT-INTO command. The keyword 'FROM' is mandatory that identifies the table name from which the data needs to be fetched.
* 'WHERE' clause is optional. If this clause is not given, then the data from the entire table will be fetched.

Cursors:

When a SQL statement is executed in PL/SQL , Oracle assigns a private work area for that statement. The work area is called cursor. This stores the statement and results returned by the execution of that statement. There are two types of cursors namely implicit and explicit.

Types of cursors

* 1. **Static curso**r: the contents of this cursor are known at compile time. The cursor object of such an SQL statement is always based on one SQL statement.
  2. **Dynamic Cursor**: a cursor variable that can change its value is used. The variable can refer to different SQL statements at different times.

**Implicit cursor**:

Pl/SQL creates an implicit cursor when an SQL is executed from within the program block. When a program is executed the Oracle opens an implicit cursor and the pointer is set to the first row in the cursor.

|  |
| --- |
| --implicit cursor example  declare  no\_rows number(2);  begin  update works\_csc  set salary=salary+(salary\*(10/100));  if sql%notfound then  dbms\_output.put\_line('no employees found');  else  no\_rows:=sql%rowcount;  dbms\_output.put\_line(no\_rows ||' employees salary updated');  end if;  end;  / |

Before execution of the above program the table has

|  |
| --- |
| SQL> select \* from works\_csc;  EID CID SALARY  ---------- ---- ----------  100 c1 45000  101 c2 35000  102 c3 35000  103 c4 50000  104 c2 30000  105 c3 30000  106 c1 40000  108 c3 30000  109 c3 28000  9 rows selected. |

After execution of the above program the table is updated as:

Output

|  |
| --- |
| SQL> select \* from works\_csc;  EID CID SALARY  ---------- ---- ----------  100 c1 49500  101 c2 38500  102 c3 38500  103 c4 55000  104 c2 33000  105 c3 33000  106 c1 44000  108 c3 33000  109 c3 30800  9 rows selected. |

The implicit cursor attributes are

|  |  |
| --- | --- |
| Attributes | Functions |
| SQL%ISOPEN | It returns TRUE if the cursor is open; otherwise it returns FALSE |
| SQL %FOUND | It returns TRUE if the last fetch returned a row, otherwise it returns FALSE |
| SQL %NOTFOUND | It returns TRUE if the last fetch did not return a row; otherwise, it returns FALSE. It complements the %FOUND attribute |
| SQL %ROWCOUNT | It returns total number of rows returned |

Explicit Cursor

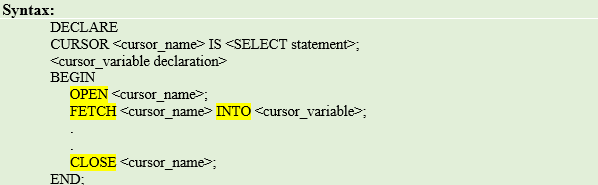
An explicit cursor is declared as a SELECT statement in PL/SQL block. The actions to be performed for an explicit cursor are:

* 1. Declare it
  2. Open it
  3. Fetch row(s) from it
  4. Close it

**Declaring an Explicit cursor**

A cursor is declared as a SELECT statement. This must not have INTO clause. If the rows are to be retrieved from table in a specific order, ORDER BY clause can be used.

Syntax is:

[](https://cdn.guru99.com/images/PL-SQL/110215_1042_SQLinPLSQL8.png)

**Syntax Explanation:**

* In the above syntax, the declaration part contains the declaration of the cursor and the cursor variable in which the fetched data will be assigned.
* The cursor is created for the 'SELECT' statement that is given in the cursor declaration.
* In execution part, the declared cursor is opened, fetched and closed.

In the following example all the rows from employee table is extracted and put it in the work area or cursor called employee\_cur

|  |
| --- |
| Declare  cursor employee\_cur  is  select \* from employee;  begin  null;  end;  / |

**Opening a cursor**:

When a cursor is opened, its select query is executed. In order to extract from work area(cursor) it must be opened. The cursor points to the first row in the active set.

**Syntax**

OPEN cursorname;

In the above example it must be:

OPEN employee\_cur;

**Fetching data from the cursor**:

The data stored in cursor can be fetched and placed in the local variables to perform further processing. After reading the first record pointed by cursor into temporary variables, now the cursor points to the next record in the workarea

The general syntax is:

FETCH cursorname INTO variablelist / recordname;

For example,

FETCH employee\_cur INTO n\_eid, v\_empname,v\_street,v\_city;

Or

FETCH employee\_cur INTO emp\_rec;

Where emp\_rec is reclared with %ROWTYPE, i.e

Emp\_rec employee\_cur%ROWTYPE;

**Closing a cursor:**

When the work is done, the cursor must be closed.

Syntax

CLOSE cursorname;

Explicit cursor attributes:

Actions can be performed on cursors with OPEN, FETCH and CLOSE statements. The attributes of cursor are:

|  |  |
| --- | --- |
| Attributes | Functions |
| %ISOPEN | It returns TRUE if the cursor is open; otherwise it returns FALSE |
| %FOUND | It returns TRUE if the last fetch returned a row, otherwise it returns FALSE |
| %NOTFOUND | It returns TRUE if the last fetch did not return a row; otherwise, it returns FALSE. It complements the %FOUND attribute |
| %ROWCOUNT | It returns total number of rows returned |

In this example the data from employee table is extracted and placed in cursor employee\_cur. The cursor is executed by the statement open cursorname. Here before opening it, the cursor is checked. The condition is checked in If clause and if it is not open it is opened or the condition moves to the loop.

Here using fetch the data from cursor is moved to the temporary variables declared and they are displayed using the following statement;

dbms\_output.put\_line(n\_eid||' '||v\_empname||' '||v\_street||' '||v\_city);

since more than one row is fetched the cursor is checked for end of the cursor or no row found by the statement employee\_cur%found as follows:

exit when not employee\_cur%found;

so cursor points to next, next rows until it reaches the end of rowset and when there is no more row to be extracted the loop is exited.

The %rowcount is used to count the rows in cursor. This is done in the statement:

dbms\_output.put\_line(employee\_cur%rowcount||' employee(s) found');

|  |
| --- |
| Declare  n\_eid employee.eid%type;  v\_empname employee.empname%type;  v\_street employee.street%type;  v\_city employee.city%type;  cursor employee\_cur  is  select \* from employee;  begin  if not employee\_cur%isopen then  open employee\_cur;  end if;  loop  fetch employee\_cur  into n\_eid,v\_empname,v\_street,v\_city;  exit when not employee\_cur%found;  dbms\_output.put\_line(n\_eid||' '||v\_empname||' '||v\_street||' '||v\_city);  end loop;  dbms\_output.put\_line(employee\_cur%rowcount||' employee(s) found');  end;  / |

Output:

|  |
| --- |
| SQL> @e:/books/sql\_prgs/cursor1.sql  100 anitha 1st street calcutta  101 aiswarya 2nd street chennai  102 chandra 2nd street chennai  103 hema 3rd street chennai  104 lalitha metha street mumbai  105 raman krishnan street bangalore  106 harini kalam street andhra  107 danush ragav street bangalore  108 david kamaraj street calcutta  109 ananthi rajaji street chennai  10 employee(s) found  PL/SQL procedure successfully completed. |

If the cursor is changed like this with a condition

|  |
| --- |
| Declare  n\_eid employee.eid%type;  v\_empname employee.empname%type;  v\_street employee.street%type;  v\_city employee.city%type;  cursor employee\_cur  is  select \* from employee  where eid=100;  begin  if not employee\_cur%isopen then  open employee\_cur;  end if;  loop  fetch employee\_cur  into n\_eid,v\_empname,v\_street,v\_city;  exit when not employee\_cur%found;  dbms\_output.put\_line(n\_eid||' '||v\_empname||' '||v\_street||' '||v\_city);  end loop;  dbms\_output.put\_line(employee\_cur%rowcount||' employee(s) found');  end;  / |

The output is

|  |
| --- |
| SQL> @e:/books/sql\_prgs/cursor1.sql  100 anitha 1st street calcutta  1 employee(s) found  PL/SQL procedure successfully completed. |

Cursor for loops

In the cursor for loop the cursor is opened implicitly when the loop starts. A row is then fetched into the record from the cursor with every iteration of the loop. The cursor is automatically closed when the loop ends, and the loop ends when there are no more rows.

Syntax:

|  |
| --- |
| FOR recordname in cursorname LOOP  Loop statements;  END LOOP; |

Where recordname is the name of the record that is declared implicitly in the loop and is destroyed when the loop ends and cursorname is the name of declared explicit cursor.

|  |
| --- |
| declare  cursor employee\_cur  is  select \* from employee;  begin  for emp\_rec in employee\_cur loop  dbms\_output.put\_line(emp\_rec.eid||' '||emp\_rec.empname||' '||emp\_rec.street||' '||emp\_rec.city);  end loop;  end;  / |

Output:

|  |
| --- |
| SQL> @e:/books/sql\_prgs/cursor2.sql  100 anitha 1st street calcutta  101 aiswarya 2nd street chennai  102 chandra 2nd street chennai  103 hema 3rd street chennai  104 lalitha metha street mumbai  105 raman krishnan street bangalore  106 harini kalam street andhra  107 danush ragav street bangalore  108 david kamaraj street calcutta  109 ananthi rajaji street chennai  PL/SQL procedure successfully completed. |

In the above program there is no fetch statement. The open cursor statement is not there. There is no checking of end of row (%FOUND) statement. All these are done by for loop. This automatically open the cursor and place the records from cursor into a record type variable and iterate and reaches end of loop when all the rows in the cursor has been gone through.

In the below program inside the for loop a condition is given where the records where the column city is having the value ‘chennai’ only is displayed.

|  |
| --- |
| declare  cursor employee\_cur  is  select \* from employee;  begin  for emp\_rec in employee\_cur loop  if emp\_rec.city='chennai' then  dbms\_output.put\_line(emp\_rec.eid||' '||emp\_rec.empname||' '||emp\_rec.street||' '||emp\_rec.city);  end if;  end loop;  end;  / |

Output:

|  |
| --- |
| SQL> @e:/books/sql\_prgs/cursor2.sql  101 aiswarya 2nd street chennai  102 chandra 2nd street chennai  103 hema 3rd street chennai  109 ananthi rajaji street chennai  PL/SQL procedure successfully completed. |

Cursor FOR loop using a SUBQUERY

If the subquery is used in a for loop there is no need to declare explicit cursor. In this declaration is not necessary as it is declared in for loop as record type. The subquery is used and the values returned is stored in the record structure. In this case the record given is emp\_rec. here the query is asked to extract the employee detail whose eid =100.

|  |
| --- |
| -- example for loop with subquery  begin  for emp\_rec in  (select \* from employee  where eid=100) loop  dbms\_output.put\_line(emp\_rec.eid||' '||emp\_rec.empname||' '||emp\_rec.street||' '||emp\_rec.city);  end loop;  end;  / |

Output:

|  |
| --- |
| SQL> @e:/books/sql\_prgs/cursor3.sql  100 anitha 1st street calcutta  PL/SQL procedure successfully completed. |

Suppose all the record are to be extracted the subquery can be given as follows:

|  |
| --- |
| -- example for loop with subquery  begin  for emp\_rec in  (select \* from employee) loop  --if emp\_rec.city='chennai' then  dbms\_output.put\_line(emp\_rec.eid||' '||emp\_rec.empname||' '||emp\_rec.street||' '||emp\_rec.city);  --end if;  end loop;  end;  / |

Output:

|  |
| --- |
| SQL> @e:/books/sql\_prgs/cursor3.sql  100 anitha 1st street calcutta  101 aiswarya 2nd street chennai  102 chandra 2nd street chennai  103 hema 3rd street chennai  104 lalitha metha street mumbai  105 raman krishnan street bangalore  106 harini kalam street andhra  107 danush ragav street bangalore  108 david kamaraj street calcutta  109 ananthi rajaji street chennai  PL/SQL procedure successfully completed. |

Another example :

|  |
| --- |
| -- example for loop with subquery  begin  for emp\_rec in  (select \* from employee where city='chennai') loop  --if emp\_rec.city='chennai' then  dbms\_output.put\_line(emp\_rec.eid||' '||emp\_rec.empname||' '||emp\_rec.street||' '||emp\_rec.city);  --end if;  end loop;  end;  / |

Output:

|  |
| --- |
| SQL> @e:/books/sql\_prgs/cursor3.sql  101 aiswarya 2nd street chennai  102 chandra 2nd street chennai  103 hema 3rd street chennai  109 ananthi rajaji street chennai  PL/SQL procedure successfully completed. |

Cursor with parameters:

A cursor can be declared with parameters like function and procedures. These values are passed to the cursor when it is opened and they are used in the query. Values can be passed inside cursor but cannot pass any value out of cursors through parameters

Syntax:

|  |
| --- |
| CURSOR cursorname  [(parameter1 datatype, parameter2 datatype,…)]  IS  SELECT query; |

Where parameter1 etc are formal parameters passed to the cursor and datatype is any scalar data type assigned to the parameter. The parameters are assigned only datatype and not size.

|  |
| --- |
| -- example for cursor with parameters  declare  n\_eid employee.eid%type:=&eid;  v\_empname employee.empname%type;  v\_street employee.street%type;  v\_city employee.city%type;  cursor emp\_cur(empid employee.eid%type) is  select empname,street,city from employee where eid=empid;  begin  open emp\_cur(n\_eid);  dbms\_output.put\_line(' eid empname street city');  loop  fetch emp\_cur into v\_empname,v\_street,v\_city;  exit when emp\_cur%notfound;  dbms\_output.put\_line(n\_eid ||' '||v\_empname||' '||v\_street||' '||v\_city);  end loop;  close emp\_cur;  end;  / |

Output

|  |
| --- |
| SQL> @e:/books/sql\_prgs/cursor4.sql;  Enter value for eid: 101  old 2: n\_eid employee.eid%type:=&eid;  new 2: n\_eid employee.eid%type:=101;  eid empname street city  101 aiswarya 2nd street chennai  PL/SQL procedure successfully completed. |

Line 2-5 : local variable declaration

Line 6 : cursor declaration with parameter empid which is of employee.eid type. In this cursor the record which satisfies the query is extracted. The parameter empid is given values while calling.

Line 9 : the cursor is executed with the parameter. This value goes inside the cursor and the query is executed with this parameter value.

Line 10 : heading is printed

Line 11 : loop starts

Line 12 : the data in cursor is extracted and transferred to local variables

Line 13 : checks for no more data in cursor. If no data found it exits the loop else continues to fetch data from cursor.

Line 14 : the data is printed

Line 15 : end of loop

Line 16 : cursor is closed.

## Composite data types: Record, Tables and varrays

## Record Type

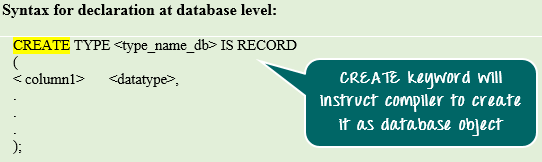
Record type is the complex and composite data type which allows the programmer to create a new data type with the desirable column structure.

Following are some of the attributes of the record type.

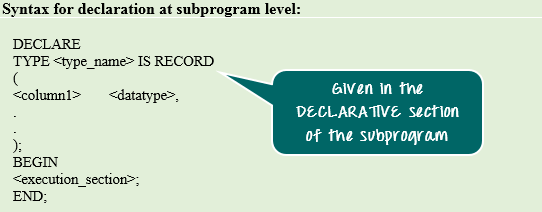
* It groups one or more column together to form a new data type
* These columns will have its own name and data type
* Record type can accept the data
  + As a single record that consists of many column OR
  + It can accept the value for one particular column of a record
* Record type simply means a new data type. Once the record type is created it will be stored as a new data type in the database and the same shall be used to declare a variable in programs.
* It will use the keyword **'TYPE'** to instruct the compiler that it is creating the new data type.
* It can be created at "**database level"** which can be stored as database objects, used all-over the database or it can be created at the "**subprogram levels"**, which is visible only inside the subprograms.
* The database level record type can also be declared for the table columns so that single column can hold the complex data.
* The data in these data type can be accessed by referring to their variable\_name followed by period operator (.) followed by column\_name i.e. '<record\_type\_variable\_name>.<column\_name>'
* Declaring a PL/SQL Record
* PL/SQL provides three ways to declare a record: *table-based record*, *cursor-based record*and *programmer-defined records.*

**Syntax for declaration at tablebase level:**

**Syntax Explanation:**

[](https://cdn.guru99.com/images/PL-SQL/110215_0744_ComplexData1.png)

* In the first syntax, we can see the keyword 'CREATE TYPE' this instructs the compiler to create the record type named "type\_name\_db" with the specified column as a database object.
* This is given as an individual statement and not inside any block.

[](https://cdn.guru99.com/images/PL-SQL/110215_0744_ComplexData2.png)

**Syntax Explanation:**

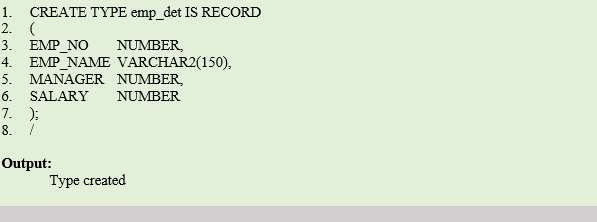
* In the syntax, we are creating the record type named "type\_name" only inside the subprogram.

In both declaration method, the way of defining the column and data type is similar.

**Example 1: RECORD Type as Database Object**

In this program, we are going to see how to create "Record type" as a database object. We are going to create record type 'emp\_det' with 4 columns. The columns and their data type are as follows:

* EMP\_NO (NUMBER)
* EMP\_NAME (VARCHAR2 (150))
* MANAGER (NUMBER)
* SALARY (NUMBER)

[](https://cdn.guru99.com/images/PL-SQL/110215_0744_ComplexData3.png)

**Code Explanation:**

* The above code will create type emp\_det as a database object.
* It will have 4 column emp\_no, emp\_name, manager and salary as defined.
* Now 'emp\_det' is a similar to other data type (like NUMBER, VARCHAR@, etc.) And it is visible in the entire database, hence this can be used in the entire database to declare the variable of this type.

**Output:**

Created the type 'emp\_det' as record type at the database level.

**Example 2: Record Type at program level- Column level access**

In this example, we are going to see how to create a record type at subprogram level and how to populate and fetch the values from it by column level.

We are going to create 'emp\_det' record\_type at subprogram level, and we are going to use the same to populate and to display data from it.

|  |
| --- |
| declare  type emp\_detail is record  (  empid number(3),  empname varchar2(30),  street varchar2(40),  city varchar2(30)  );  emp\_rec emp\_detail;  begin  emp\_rec.empid:=112;  emp\_rec.empname:='krish';  emp\_rec.street:='3rd street';  emp\_rec.city:='bangalore';  dbms\_output.put\_line('employee details');  dbms\_output.put\_line('employee id: '||emp\_rec.empid);  dbms\_output.put\_line('employee name: '||emp\_rec.empname);  dbms\_output.put\_line('employee street: '||emp\_rec.street);  dbms\_output.put\_line('employee city: '||emp\_rec.city);  end;  / |

|  |
| --- |
| SQL> @e:/books/sql\_prgs/record1.sql  employee details  employee id: 112  employee name: krish  employee street: 3rd street  employee city: bangalore  PL/SQL procedure successfully completed. |

**Code Explanation:**

* **Code line 2-8**: Record type 'emp\_det' is declared with columns empid, empname, street and city of data type NUMBER, VARCHAR2, VARCHAR2, VARCHAR2.
* **Code line 9:** emp\_rec variable is declared as 'emp\_det' data type. Now this variable can hold the value that contains all the above 4 fields/columns.
* **Code line 11:** Populating the 'emp\_id' field of 'emp\_rec' with value 112.
* **Code line 12:** Populating the 'empname' field of 'emp\_rec' with value krish.
* **Code line 13:** Populating the 'street' field of 'emp\_rec' with value 3rd street.
* **Code line 14:** Populating the 'city' field of 'emp\_rec' with value bangalore.
* **Code line 15-19:** Displaying the value of the 'emp\_rec' in output.

**Example 3: Record Type at Subprogram level-Row level access**

In this example, we are going to see how to create a record type at subprogram level and how to populate it as a row level. We are going to create 'emp\_det' record\_type at subprogram level, and we are going to use the same to populate and to display data from it.

|  |
| --- |
| declare  type emp\_detail is record  (  empid number(3),  empname varchar2(30),  street varchar2(40),  city varchar2(30)  );  emp\_rec emp\_detail;  begin  emp\_rec.empid:=112;  emp\_rec.empname:='krish';  emp\_rec.street:='3rd street';  emp\_rec.city:='bangalore';  insert into employee(eid,empname,street,city) values  (emp\_rec.empid,emp\_rec.empname,emp\_rec.street,emp\_rec.city);  commit;  select eid,empname,street,city into emp\_rec from employee where eid=102;  dbms\_output.put\_line('employee details');  dbms\_output.put\_line('employee id: '||emp\_rec.empid);  dbms\_output.put\_line('employee name: '||emp\_rec.empname);  dbms\_output.put\_line('employee street: '||emp\_rec.street);  dbms\_output.put\_line('employee city: '||emp\_rec.city);  end;  / |

OUTPUT

|  |
| --- |
| SQL> @e:/books/sql\_prgs/record2.sql  employee details  employee id: 102  employee name: chandra  employee street: 2nd street  employee city: chennai  PL/SQL procedure successfully completed. |

**Code Explanation:**

* **Code line 2-8**: Record type 'emp\_det' is declared with columns empid, empname, street and city of data type NUMBER, VARCHAR2, VARCHAR2, VARCHAR2.
* **Code line 9:** emp\_rec variable is declared as 'emp\_detail' data type. Now this variable can hold the value that contains all the above 4 fields/columns.
* **Code line 11-14:** Populating the table employee with data 112 as empid, krish as empname, 3rd street as street and bangalore as city.
* **Code line 15:** Committing the above insert transaction.
* **Code line 16:** Populating the 'emp\_rec' variable as a row level data from the select query for employee number 102.
* **Code line 17-21:** Displaying the value of the 'emp\_rec' in output.

**Note:** The record type can be accessed only in column level while redirecting its value to any output mode.

Another example

|  |
| --- |
| declare  type emp\_detail is record  (  empid employee.eid%type,  empname employee.empname%type,  street employee.street%type,  city employee.city%type  );  emp\_rec emp\_detail;  begin  select eid,empname,street,city into emp\_rec from employee where eid=103;  dbms\_output.put\_line('employee details');  dbms\_output.put\_line('employee id: '||emp\_rec.empid);  dbms\_output.put\_line('employee name: '||emp\_rec.empname);  dbms\_output.put\_line('employee street: '||emp\_rec.street);  dbms\_output.put\_line('employee city: '||emp\_rec.city);  end;  / |

Output

|  |
| --- |
| SQL> @e:/books/sql\_prgs/record3.sql  employee details  employee id: 103  employee name: hema  employee street: 3rd street  employee city: chennai  PL/SQL procedure successfully completed. |

Declaring Table-based Record

To declare a table-based record you use a table name with %ROWTYPE attribute*.*The fields of the PL/SQL record has the same name and data type as the column of the table.

The following illustrates table-based record declaration:

|  |  |
| --- | --- |
| 1  2 | DECLARE     table\_based\_record table\_name%ROWTYPE; |

After having the table-based record, you can use it in various ways, for example in SQL SELECT statement as follows:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | SET SERVEROUTPUT ON SIZE 1000000;  DECLARE    r\_empemployees%ROWTYPE;    n\_emp\_id  employees.employee\_id%TYPE :=200;  BEGIN    SELECT \*    INTO r\_emp    FROM employees    WHERE employee\_id=n\_emp\_id;    -- print out the employee's first name    DBMS\_OUTPUT.PUT\_LINE(r\_emp.first\_name);  END;  / |

In the above example:

* First, we defined a record based on employees table in database.
* Second, we used the SELECT statement to retrieve the employee information of the employee id 200 and populate the data into the r\_emp record .
* Third, we print out the first name of the selected employee from the r\_empemployee record.

Declaring Cursor-based Record

You can define a record based on a [cursor](http://www.plsqltutorial.com/plsql-cursor/). First, you must define a cursor. And then you use %ROWTYPE with the cursor variable to declare a record. The fields of the record correspond to the columns in the cursor SELECT statement.

The following is an example of declaring a record based on a cursor.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | SET SERVEROUTPUT ON SIZE 1000000;  DECLARE    CURSOR cur\_emp IS      SELECT \*      FROM employees      WHERE employee\_id=200;      emp\_reccur\_emp%ROWTYPE;  BEGIN    NULL;  END;  / |

Working with PL/SQL Record

After having a PL/SQL record, you can work with a record as a whole or you can work with individual field of the record.

Working with PL/SQL record at record level

At record level, you can do the following:

* You can assign a PL/SQL record to another PL/SQL record. The pair of PL/SQL records must have the same number of fields and the data type of each field has to be convertible.
* You can assign a PL/SQL record NULL value by assigning an uninitialized record.
* A PL/SQL record can be used as an argument of parameter in a [function](http://www.plsqltutorial.com/plsql-function/)
* You can return a PL/SQL record from a [function](http://www.plsqltutorial.com/plsql-function/)
* To check if the record is NULL, you have to check each individual field of the record.
* To compare two records, you have to compare each individual field of each record.

Here is an example of working with PL/SQL record at record level:

|  |  |
| --- | --- |
|  | SET serveroutput ON SIZE 1000000;  DECLARE  TYPE t\_name IS RECORD(  empid employee\_csc.eid%TYPE,  ename employee\_csc.ename%TYPE  );  r\_name t\_name;  r\_name2 t\_name;  r\_name\_null t\_name;  n\_emp\_id employee\_csc.eid%TYPE :=101;  BEGIN  -- assign employee's infomation to record  SELECT eid,ename  INTO r\_name  FROM employee\_csc  WHERE eid=n\_emp\_id;  -- assign record to another record  r\_name2:=r\_name;  -- print out the employee's name  DBMS\_OUTPUT.PUT\_LINE(r\_name2.empid|| ',' ||r\_name2.ename);  -- assign record to NULL  r\_name2:=r\_name\_null;  -- check NULL for each individual field  IF r\_name2.empid IS NULL AND  r\_name2.ename IS NULL THEN  DBMS\_OUTPUT.PUT\_LINE('Record r\_name2 is NULL');  END IF;  END;  / |

Output

|  |
| --- |
| SQL> @d:\sql\_prgs\record1.sql  101,aiswarya  Record r\_name2 is NULL  PL/SQL procedure successfully completed. |

Working with PL/SQL record at field level

As you see in the above example, we can reference to a field of a record by using dot notation (.) as follows:

|  |  |
| --- | --- |
| 1 | record\_name.field |

If you reference to a record variable in different package or schema you need to explicitly specify those information as shown below:

|  |  |
| --- | --- |
| 1 | [schema\_name.][package\_name.]record\_name.field |

You can use the assignment operator ( :=) to change the value of field of a record that you reference to.

For the nested record you need to use extra dot notation ( .)

The following example demonstrates how to use PL/SQL record a field level:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | DECLARE    TYPE t\_address IS RECORD  (      house\_number VARCHAR2(6),      street       VARCHAR2(50),      phone        VARCHAR2(15),      region       VARCHAR2(10),      postal\_code  VARCHAR2(10),      country      VARCHAR2(25)    );      TYPE t\_contact IS RECORD  (      home     t\_address,      businesst\_address    );    r\_contactt\_contact;  BEGIN    r\_contact.business.house\_number:= '500';    r\_contact.business.street       := 'Oracle Parkway';    r\_contact.business.region       := 'CA';    r\_contact.business.postal\_code  := '94065';    r\_contact.business.country      := 'USA';    r\_contact.business.phone        := '+1.800.223.1711';  END; |

PL/SQL COLLECTIONS

The collection is nothing but an ordered group of elements of particular data types. It can be a collection of simple data type or complex data type (like user defined or record types).

In the collection, each element is identified by a term called **"subscript."** Each item in the collection is assigned with a unique subscript. The data in that collection can be manipulated or fetched by referring to that unique subscript.

Collections are most useful things when a large data of the same type needs to be processed or manipulated. Collections can be populated and manipulated as whole using 'BULK' option in Oracle.

Collections are classified based on the structure, subscript and storage as shown below.

* Index-by-tables (also known as Associative Array)
* Nested tables
* Varrays

At any point, data in the collection can be referred by three terms Collection name, Subscript, Field/Column name as "<collection\_name>(<subscript>).<column\_name>".

## PL/SQL Tables

A PL/SQL table is a single-dimensional structure which stores homogenous data.

Declaring a PL/SQL table can be done in two steps:

1. declare a PL/SQL table type with a TYPE statement.

2. declare an actual table based on the type declared in the previous step.

## Index-By Table

An **index-by** table (also called an associative array) is a set of **key-value** pairs. Each key is unique and is used to locate the corresponding value. The key can be either an integer or a string.

Index-by-table is a collection in which the array size is not fixed. Unlike the other collection types, in the index-by-table collection the subscript can be defined by the user. Following are the attributes of index-by-table.

* The subscript can of integer or strings. At the time of creating the collection, the subscript type should be mentioned.
* These collections are not stored sequentially.
* They are always sparse in nature.
* The array size is not fixed.
* They cannot be stored in the database column. They shall be created and used in any program in that particular session.
* They give more flexibility in terms of maintaining subscript.
* The subscripts can be of negative subscript sequence also.
* They are more appropriate to use for relatively smaller collective values in which the collection can be initialized and used within the same subprograms.
* They need not to be initialized before start using them.
* It cannot be created as a database object. It can only be created inside the subprogram, which can be used only in that subprogram.
* BULK COLLECT cannot be used in this collection type as the subscript should be given explicitly for each record in the collection.

An index-by table is created using the following syntax. Here, we are creating an index-by table named **table\_name** whose keys will be of *subscript\_type* and associated values will be of *element\_type*

Syntax

|  |
| --- |
| TYPE type\_name IS TABLE OF element\_type [NOT NULL] INDEX BY subscript\_type;  table\_name type\_name; |

example

|  |
| --- |
| Type salary is table of number index by varchar2(20);  Salary\_list salary; |

In the above example a table by name salary is created which stores integer values whose index will consists of string values.

In the next statement salary\_list is a variable of type salary table.

|  |
| --- |
| declare  **type salary is table of number index by varchar2(20);**  **salary\_list salary;**  name varchar2(20);  begin  -- adding elements to table  salary\_list('anitha'):=3000;  salary\_list('aswin'):=5000;  salary\_list('harini'):=4000;  salary\_list('banu'):=10000;  salary\_list('guru'):=5000;  -- printing table  name:=salary\_list.first;  while name is not null loop  dbms\_output.put\_line('salary of '|| name ||' is '||salary\_list(name));  name:=salary\_list.next(name);  end loop;  end;  / |

Output

|  |
| --- |
| SQL> @e:/rdbms/plsql/tableeg1.sql  salary of anitha is 3000  salary of aswin is 5000  salary of banu is 10000  salary of guru is 5000  salary of harini is 4000 |

## Example:

Elements of an index-by table could also be a %ROWTYPE of any database table or %TYPE of any database table field.

|  |
| --- |
| declare  cursor emp\_cur is  select empname from employee;  **type e\_list is table of employee.empname%type index by binary\_integer;**  **emp\_list e\_list;**  counter integer:=0;  begin  for temp in emp\_cur loop  counter:=counter+1;  emp\_list(counter):=temp.empname;  dbms\_output.put\_line('employee name: '|| emp\_list(counter));  end loop;  end;  / |

Output:

|  |
| --- |
| SQL> @e:/rdbms/plsql/tableeg2.sql;  employee name: anitha  employee name: aiswarya  employee name: chandra  employee name: hema  employee name: lalitha  employee name: raman  employee name: harini  employee name: danush  employee name: david  employee name: ananthi  employee name: RAJAN  employee name: krish  PL/SQL procedure successfully completed. |

## Nested Tables

A **nested table** is like a one-dimensional array with an arbitrary number of elements. However, a nested table differs from an array in the following aspects:

* An array has a declared number of elements, but a nested table does not. The size of a nested table can increase dynamically.
* An array is always dense, i.e., it always has consecutive subscripts. A nested array is dense initially, but it can become sparse when elements are deleted from it.

A **nested table** is created using the following syntax:

**TYPE type\_name IS TABLE OF element\_type [NOT NULL];**

**table\_name type\_name;**

This declaration is similar to declaration of an **index-by** table, but there is no INDEX BY clause.

A nested table can be stored in a database column and so it could be used for simplifying SQL operations where you join a single-column table with a larger table. An associative array cannot be stored in the database.

Example

|  |
| --- |
| declare  **type nametable is table of varchar2(20);**  **type grade is table of integer;**  names nametable;  mark grade;  total integer;  begin  names:=nametable('anitha','aswini','kamala','keerthana','malathi');  mark:=grade(56,89,78,70,66);  total:=names.count;  dbms\_output.put\_line('total:'|| total ||' students');  for i in 1..total loop  dbms\_output.put\_line('student: '||names(i) ||' marks: '||mark(i));  end loop;  end;  / |

Output

|  |
| --- |
| SQL> @e:/rdbms/plsql/tableeg3.sql;  total:5 students  student: anitha marks: 56  student: aswini marks: 89  student: kamala marks: 78  student: keerthana marks: 70  student: malathi marks: 66  PL/SQL procedure successfully completed. |

## Example 2

Elements of a **nested table** could also be a %ROWTYPE of any database table or %TYPE of any database table field. The following example illustrates the concept. We will use the EMPLOYEE table stored in our database as:

|  |
| --- |
| declare  cursor emp\_cur is  select empname from employee;  **type emp\_table is table of employee.empname%type;**  **emplist emp\_table:=emp\_table();**  counter integer:=0;  begin  for temp in emp\_cur loop  counter:=counter+1;  emplist.extend;  emplist(counter):=temp.empname;  dbms\_output.put\_line('employee name: '||emplist(counter) );  end loop;  dbms\_output.put\_line('number of employees are : '||counter );  end;  / |

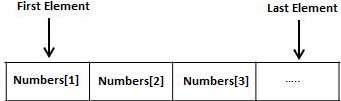
Output

|  |
| --- |
| SQL> @e:/rdbms/plsql/tableeg4.sql;  employee name: anitha  employee name: aiswarya  employee name: chandra  employee name: hema  employee name: lalitha  employee name: raman  employee name: harini  employee name: danush  employee name: david  employee name: ananthi  employee name: RAJAN  employee name: krish  number of employees are : 12  PL/SQL procedure successfully completed. |

PL/SQL Varrays

This is a composite data type and stands for variable-size array. This is single-dimensional, bounded collections of elements of same data type.

Varray is a collection method in which the size of the array is fixed. The array size cannot be exceeded than its fixed value. The subscript of the Varray is of a numeric value. Following are the attributes of Varrays.

* Upper limit size is fixed
* Populated sequentially starting with the subscript '1'
* This collection type is always dense, i.e. we cannot delete any array elements. Varray can be deleted as a whole, or it can be trimmed from the end.
* Since it always be dense in nature, it has very less flexibility.
* It is more appropriate to use when the array size is known and to perform a similar activities on all the array elements.
* The subscript and sequence always remain stable, i.e. the subscript and count of the collection is always same.
* They needs to be initialized before using them in programs. Any operation (except EXISTS operation) on uninitialized collection will throw an error.
* It can be created as a database object, which is visible throughout the database or inside the subprogram, which can be used only in that subprogram.
* All varrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.
* 

**Creating a Varray Type**

A varray type is created with the CREATE TYPE statement. Maximum size and the type of elements stored in the varray must be specified.

The basic syntax for creating a VRRAY type at the schema level is:

**CREATE OR REPLACE TYPE varray\_type\_name IS VARRAY(n) of <element\_type>**

Where,

* *varray\_type\_name* is a valid attribute name,
* *n* is the number of elements (maximum) in the varray,
* *element\_type* is the data type of the elements of the array.

Maximum size of a varray can be changed using the ALTER TYPE statement.

For example,

**CREATE Or REPLACE TYPE namearray AS VARRAY(3) OF VARCHAR2(10);**

/

Type created.

The basic syntax for creating a VARRAY type within a PL/SQL block is:

The general syntax is:

**TYPE varray\_type\_name IS VARRAY(size) of <element\_type>[NOT NULL];**

**Varrayname varray\_type\_name;**

For example:

**TYPE namearray IS VARRAY(5) OF VARCHAR2(10);**

**Type grades IS VARRAY(5) OF INTEGER;**

Example:

**TYPE emparray\_type is varray(5) of employee\_csc.eid%type;**

**eid\_varray emparray\_type:=emparray\_type();**

* In oracle environment, the starting index for varrays is always 1.
* Varrays are one-dimensional arrays.
* A varray is automatically NULL when it is declared and must be initialized before its elements can be referenced

|  |
| --- |
| declare  **type namearray is varray(5) of varchar2(20);**  **type grade is varray(5) of integer;**  **names namearray;**  **mark grade;**  total integer;  begin  names:=namearray('anitha','aswini','kamala','keerthana','malathi');  mark:=grade(56,89,78,70,66);  total:=names.count;  dbms\_output.put\_line('total:'|| total ||' students');  for i in 1..total loop  dbms\_output.put\_line('student: '||names(i) ||' marks: '||mark(i));  end loop;  end;  / |

Output

|  |
| --- |
| SQL> @e:/rdbms/plsql/tableeg5.sql;  total:5 students  student: anitha marks: 56  student: aswini marks: 89  student: kamala marks: 78  student: keerthana marks: 70  student: malathi marks: 66  PL/SQL procedure successfully completed. |

## Example 2

Elements of a varray could also be a %ROWTYPE of any database table or %TYPE of any database table field. The following example illustrates the concept:

|  |
| --- |
| declare  cursor emp\_cur is  select empname from employee;  **type emp\_array is varray(15) of employee.empname%type;**  **emplist emp\_array:=emp\_array();**  counter integer:=0;  begin  for temp in emp\_cur loop  counter:=counter+1;  emplist.extend;  emplist(counter):=temp.empname;  dbms\_output.put\_line('employee name: '||emplist(counter) );  end loop;  end;  / |

Output

|  |
| --- |
| SQL> @e:/rdbms/plsql/tableeg6.sql;  employee name: anitha  employee name: aiswarya  employee name: chandra  employee name: hema  employee name: lalitha  employee name: raman  employee name: harini  employee name: danush  employee name: david  employee name: ananthi  employee name: RAJAN  employee name: krish  PL/SQL procedure successfully completed. |

## Constructor and Initialization Concept in Collections

Constructors are the in-built function provided by the oracle that has the same name as of the object or collections. They are executed first whenever object or collections are getting referred for the first time in a session. Below are the important details of constructor in collection context:

* For collections, these constructors should be called explicitly to initialize it.
* Both Varray and Nested tables need to be initialized through these constructors before getting referred into the program.
* Constructor implicitly extends the memory allocation for a collection (except Varray), hence constructor can also assign the variables to the collections.
* Assigning values to the collection through constructors will never make the collection sparse.

## Collection Methods

Oracle provide many functions to manipulate and to work with the collections. These functions are very much useful in the program to determine and to modify the different attribute of the collections. Following table will give the different functions and their description.

|  |  |  |
| --- | --- | --- |
| **Method** | **Description** | **SYNTAX** |
| EXISTS (n) | This method will return Boolean results. It will return 'TRUE' if the nth element exists in that collection, else it will return FALSE. Only EXISTS functions can be used in uninitialized collection | <collection\_name>.EXISTS(element\_position) |
| COUNT | Gives the total count of the elements present in a collection | <collection\_name>.COUNT |
| LIMIT | It returns the maximum size of the collection. For Varray, it will return the fixed size that has been defined. For Nested table and Index-by-table, it gives NULL | <collection\_name>.LIMIT |
| FIRST | Returns the value of the first index variable(subscript) of the collections | <collection\_name>.FIRST |
| LAST | Returns the value of the last index variable(subscript) of the collections | <collection\_name>.LAST |
| PRIOR (n) | Returns precedes index variable in a collection of the nth element. If there is no precedes index value NULL is returned | <collection\_name>.PRIOR(n) |
| NEXT (n) | Returns succeeds index variable in a collection of the nth element. If there is no succeeds index value NULL is returned | <collection\_name>.NEXT(n) |
| EXTEND | Extends one element in a collection at the end | <collection\_name>.EXTEND |
| EXTEND (n) | Extends n elements at the end of a collection | <collection\_name>.EXTEND(n) |
| EXTEND (n,i) | Extends n copies of the ith element at the end of the collection | <collection\_name>.EXTEND(n,i) |
| TRIM | Removes one element from the end of the collection | <collection\_name>.TRIM |
| TRIM (n) | Removes n elements from the end of collection | <collection\_name>.TRIM (n) |
| DELETE | Deletes all the elements from the collection. Makes the collection empty | <collection\_name>.DELETE |
| DELETE (n) | Deletes the nth element from the collection. If the nth element is NULL, then this will do nothing | <collection\_name>.DELETE(n) |
| DELETE (m,n) | Deletes the element in the range mth to nth in the collection | <collection\_name>.DELETE(m,n) |

# PL/SQL Exception Handling

## What is Exception

An error occurs during the program execution is called Exception in PL/SQL.

PL/SQL facilitates programmers to catch such conditions using exception block in the program and an appropriate action is taken against the error condition.

There are two type of exceptions:

* System-defined Exceptions
* User-defined Exceptions

## PL/SQL Exception Handling

**Syntax for exception handling:**

Following is a general syntax for exception handling:

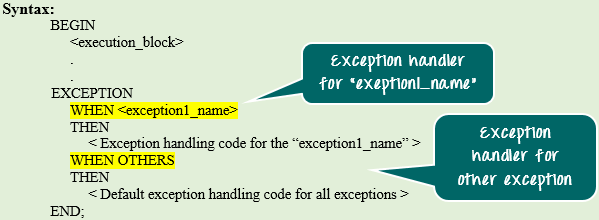
1. DECLARE
2. <declarations section>
3. BEGIN
4. <executable command(s)>
5. EXCEPTION
6. <exception handling goes here >
7. WHEN exception1 THEN
8. exception1-handling-statements
9. WHEN exception2  THEN
10. exception2-handling-statements
11. WHEN exception3 THEN
12. exception3-handling-statements
13. ........
14. WHEN others THEN
15. exception3-handling-statements
16. END;

## Exception-Handling

In real time processes often such exceptions are unavoidable. These errors will stop the program from executing further, so in order to avoid such condition they need to be captured and handled separately. This process is called as Exception-Handling, in which the programmer handles the exception that can possibly occur at the run time.

These exceptions are handled at the block level, i.e. once if any exception occurs in any block then the control will come out of execution part of that block. The exception will then be handled at the exception handling part of that block. After handling the exception, it is not possible to resend control back to the execution section of that block.

The below syntax explains how to catch and handle the exception.

[](https://cdn.guru99.com/images/PL-SQL/110215_1024_ExceptionHa1.png)

**Syntax Explanation:**

* In the above syntax, the exception-handling block contains series of WHEN condition to handle the exception.
* Each WHEN condition is followed by the exception name which is expected to be raised at the run time.
* When any exception is raised at runtime then the PL/SQL engine will look in the exception handling part for that particular exception. It will start from the first 'WHEN' clause and sequentially it will search.
* If it found the exception handling for the exception which has been raised, then it will execute that particular handling code part.
* If none of the 'WHEN' clause is present for the exception which has been raised, then PL/SQL engine will execute the 'WHEN OTHERS' part (if present). This is the common for all the exception.
* After executing the exception part control will go out of the current block.
* Only one exception part can be executed for a block at run-time. After executing it, the controller will skip the remaining exception handling part and will go out of the current block.

**Note:** WHEN OTHERS should always be at the last position of the sequence. The exception handling part present after WHEN OTHERS will never get executed as the control will exit from the block after executing the WHEN OTHERS.

## Predefined Exceptions

Oracle has predefined some common exception. These exceptions have a unique exception name and error number. These exceptions are already defined in the 'STANDARD' package in oracle. In code, we can directly use these predefined exception name to handle them.

Below are the few predefined exceptions

|  |  |  |
| --- | --- | --- |
| **Exception** | **Error Code** | **Exception Reason** |
| ACCESS\_INTO\_NULL | ORA-06530 | Assign value to the attributes of uninitialized objects |
| CASE\_NOT\_FOUND | ORA-06592 | None of the 'WHEN' clause in CASE statement satisfied and no 'ELSE' clause is specified |
| COLLECTION\_IS\_NULL | ORA-06531 | Using collection methods (except EXISTS) or accessing collection attributes on a uninitialized collections |
| CURSOR\_ALREADY\_OPEN | ORA-06511 | Trying to open a cursor which is already opened |
| DUP\_VAL\_ON\_INDEX | ORA-00001 | Storing duplicate value in a database column that is constrained by unique index |
| INVALID\_CURSOR | ORA-01001 | Illegal cursor operations like closing an unopened cursor |
| INVALID\_NUMBER | ORA-01722 | Conversion of character to number failed due to invalid number character |
| NO\_DATA\_FOUND | ORA-01403 | When 'SELECT' statement that contains INTO clause fetches no rows. |
| ROW\_MISMATCH | ORA-06504 | When cursor variable data type is incompatible with the actual cursor return type |
| SUBSCRIPT\_BEYOND\_COUNT | ORA-06533 | Referring collection by index number that is larger than the collection size |
| SUBSCRIPT\_OUTSIDE\_LIMIT | ORA-06532 | Referring collection by index number that is outside the legal range (eg: -1) |
| TOO\_MANY\_ROWS | ORA-01422 | When a 'SELECT' statement with INTO clause returns more than one row |
| VALUE\_ERROR | ORA-06502 | Arithmetic or size constraint error (eg: assigning a value to a variable that is larger than the variable size) |
| ZERO\_DIVIDE | ORA-01476 | Dividing a number by '0' |

The below program doesn’t handle exception.

Program for divide by zero error

|  |
| --- |
| declare  a number(2):=&a;  b number(2):=&b;  c number(4,2);  begin  c:=a/b;  dbms\_output.put\_line(a||'/'||b||' = '||c);  end;  / |

Output: without error handling

|  |
| --- |
| SQL> /  Enter value for a: 5  old 2: a number(2):=&a;  new 2: a number(2):=5;  Enter value for b: 0  old 3: b number(2):=&b;  new 3: b number(2):=0;  declare  \*  ERROR at line 1:  ORA-01476: divisor is equal to zero  ORA-06512: at line 6 |

in the below program under exception section, when a divide by zero error occurs what should be done is given.

With proper handling of exception

|  |
| --- |
| declare  a number(2):=&a;  b number(2):=&b;  c number(4,2);  begin  c:=a/b;  dbms\_output.put\_line(a||'/'||b||' = '||c);  **exception**  **when zero\_divide then**  dbms\_output.put\_line('you entered 0 for divider. enter some valid numerical value');  end;  / |

Output: after handling of exception

|  |
| --- |
| SQL> @e:/books/sql\_prgs/divide\_exp1.sql;  Enter value for a: 6  old 2: a number(2):=&a;  new 2: a number(2):=6;  Enter value for b: 0  old 3: b number(2):=&b;  new 3: b number(2):=0;  you entered 0 for divider. enter some valid numerical value  PL/SQL procedure successfully completed. |

Exception

|  |
| --- |
| Declare  n\_eid employee.eid%type;  v\_empname employee.empname%type;  v\_street employee.street%type;  v\_city employee.city%type;  begin  select eid,empname,street,city  into n\_eid,v\_empname,v\_street,v\_city  from employee;  dbms\_output.put\_line('emp id '||n\_eid);  dbms\_output.put\_line('emp name '||v\_empname);  dbms\_output.put\_line('emp street '||v\_street);  dbms\_output.put\_line('city '||v\_city);  end;  / |

In this the select statement where condition is not given. It returns more than one row. So the following error occurs:

|  |
| --- |
| SQL> @e:/books/sql\_prgs/selectinto1.sql  declare  \*  ERROR at line 1:  ORA-01422: exact fetch returns more than requested number of rows  ORA-06512: at line 7 |

In this below example the exception part handles the error.

|  |
| --- |
| declare  n\_eid employee.eid%type;  v\_empname employee.empname%type;  v\_street employee.street%type;  v\_city employee.city%type;  begin  select eid,empname,street,city  into n\_eid,v\_empname,v\_street,v\_city  from employee;  dbms\_output.put\_line('emp id '||n\_eid);  dbms\_output.put\_line('emp name '||v\_empname);  dbms\_output.put\_line('emp street '||v\_street);  dbms\_output.put\_line('city '||v\_city);  **exception**  when **no\_data\_found** then  dbms\_output.put\_line('no such employee exists');  when **too\_many\_rows** then  dbms\_output.put\_line('more than one row returned');  end;  / |

|  |
| --- |
| SQL> @e:/books/sql\_prgs/selectinto\_exception1.sql  more than one row returned  PL/SQL procedure successfully completed. |

## Example of exception handling

Let's take a simple example to demonstrate the concept of exception handling. Here we are using the already created CUSTOMERS table.

SELECT\* FROM COUSTOMERS;

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **NAME** | **AGE** | **ADDRESS** | **SALARY** |
| 1 | Ramesh | 23 | Allahabad | 20000 |
| 2 | Suresh | 22 | Kanpur | 22000 |
| 3 | Mahesh | 24 | Ghaziabad | 24000 |
| 4 | Chandan | 25 | Noida | 26000 |
| 5 | Alex | 21 | Paris | 28000 |
| 6 | Sunita | 20 | Delhi | 30000 |

1. DECLARE
2. c\_id customers.id%type := 8;
3. c\_name  customers.name%type;
4. c\_addr customers.address%type;
5. BEGIN
6. SELECT  name, address INTO  c\_name, c\_addr
7. FROM customers
8. WHERE id = c\_id;
9. DBMS\_OUTPUT.PUT\_LINE ('Name: '||  c\_name);
10. DBMS\_OUTPUT.PUT\_LINE ('Address: ' || c\_addr);
11. EXCEPTION
12. WHEN **no\_data\_found** THEN
13. dbms\_output.put\_line('No such customer!');
14. WHEN **others** THEN
15. dbms\_output.put\_line('Error!');
16. END;
17. /

After the execution of above code at SQL Prompt, it produces the following result:

No such customer!

PL/SQL procedure successfully completed.

The above program should show the name and address of a customer as result whose ID is given. But there is no customer with ID value 8 in our database, so the program raises the run-time exception NO\_DATA\_FOUND, which is captured in EXCEPTION block.

#### Note: You get the result "No such customer" because the customer\_id used in the above example is 8 and there is no cutomer having id value 8 in that table.

If you use the id defined in the above table (i.e. 1 to 6), you will get a certain result. For a demo example: here, we are using the id 5.

1. DECLARE
2. c\_id customers.id%type := 5;
3. c\_name  customers.name%type;
4. c\_addr customers.address%type;
5. BEGIN
6. SELECT  name, address INTO  c\_name, c\_addr
7. FROM customers
8. WHERE id = c\_id;
9. DBMS\_OUTPUT.PUT\_LINE ('Name: '||  c\_name);
10. DBMS\_OUTPUT.PUT\_LINE ('Address: ' || c\_addr);
11. EXCEPTION
12. WHEN **no\_data\_found** THEN
13. dbms\_output.put\_line('No such customer!');
14. WHEN **others** THEN
15. dbms\_output.put\_line('Error!');
16. END;
17. /

After the execution of above code at SQL prompt, you will get the following result:

Name: alex

Address: paris

PL/SQL procedure successfully completed.

## Non-predefined oracle server exceptions

A nonpredefined oracle server exception has an attached Oracle error code. This can be trapped with a WHEN OTHERS clause or by declaring them with names in the DECLARE section

**Pragma Exception\_Init**

You can define your own error message and error number using Pragma EXCEPTION\_INIT or RAISE\_APPLICATION\_ERROR function. PRAGMA is a compiler directive that associates an exception name with an internal Oracle error code. The PRAGMA directive is not processed with the execution of a PL/SQL block, but it directs the PL/SQL compiler to associate a name with the error code.

Often the programs need to handle an Oracle error that has a particular number associated with it, but no name by which it can be referenced. As a result, it becomes unable to write a handler and trap this error. In such cases the error numbers can be given a name using pragma.

Naming and associating are two separate statements in the declaration section. An exception name has to be declared as an exception and the declared name is associated with an internal error code returned by SQLCODE with the PRAGMA directive. The general syntax is :

Exceptionname Exception;

PRAGMA EXCEPTION\_INIT (exceptionname, errornumber);

Where exceptionname is user supplied and errornumber is oracle’s internal error code.

Syntax:

|  |
| --- |
| DECLARE  user\_define\_exception\_name EXCEPTION;  PRAGMA EXCEPTION\_INIT(user\_define\_exception\_name,-error\_number);  BEGIN  statement(s);  IF condition THEN  RAISE user\_define\_exception\_name;  ENDIF;  EXCEPTION  WHEN user\_define\_exception\_name THEN  User defined statement (action) will be taken;  END; |

Example

|  |
| --- |
| declare  myex exception;  PRAGMA EXCEPTION\_INIT(myex,-20015);  n number:=&n;  begin  for i in 1..n loop  dbms\_output.put\_line(i);  if i=n then  raise myex;  end if;  end loop;  exception  when myex then  dbms\_output.put\_line('loop finish');  end;  / |

Output

|  |
| --- |
| SQL> @e:/plsql/pragmaexp1.sql  Enter value for n: 5  old 4: n number:=&n;  new 4: n number:=5;  1  2  3  4  5  loop finish  PL/SQL procedure successfully completed. |

**Example 2:**

|  |
| --- |
| declare  begin  insert into employee values('','raman','kalam street','chennai');  commit;  dbms\_output.put\_line('one record is added');  end;  / |

When the above code is executed the following error occurs as a null value is tried to insert which is not allowed.

Output

|  |
| --- |
| SQL> @e:/plsql/pragmaexp2.sql;  declare  \*  ERROR at line 1:  ORA-01400: cannot insert NULL into ("SYSTEM"."EMPLOYEE"."EMP\_ID")  ORA-06512: at line 3 |

**Now knowing the error code (ORA-01400) this can be used with pragma exception\_init to give a name and write our own error message like below:**

|  |
| --- |
| declare  e\_constraint\_violation exception;  PRAGMA EXCEPTION\_INIT(e\_constraint\_violation, -1400);  begin  insert into employee values('','raman','kalam street','chennai');  commit;  dbms\_output.put\_line('one record is added');  exception  when e\_constraint\_violation then  dbms\_output.put\_line('insert statement null values are inserted which is not allowed');  end;  / |

**OUTPUT**

|  |
| --- |
| SQL> @e:/plsql/pragmaexp2.sql;  insert statement null values are inserted which is not allowed  PL/SQL procedure successfully completed. |

**Example 3:**

|  |
| --- |
| **SQL> DELETE FROM EMPLOYEE WHERE EMP\_ID='e01';**  **DELETE FROM EMPLOYEE WHERE EMP\_ID='e01'**  **\***  **ERROR at line 1:**  **ORA-02292: integrity constraint (SYSTEM.SYS\_C007000) violated - child record**  **found** |

**When the above code is executed in the sql prompt we will get to know the error code as -2292. This can be used inside pl/sql program to assign a name for it as follows**

|  |
| --- |
| declare  emp\_del exception;  PRAGMA EXCEPTION\_INIT(emp\_del, -2292);  begin  delete from employee where emp\_id='&eid';  commit;  dbms\_output.put\_line('one record is deleted');  exception  when emp\_del then  dbms\_output.put\_line('cannot be removed as child is present in another table');  end;  / |

In the above example another table named company has emp\_id column as foreign key which is referencing the emp\_id as primary key in employee table.

When the record in teh employee table which has its child record in company tries to be deleted, an error is raised. The user can give proper error message to handle this error.

**Output**

|  |
| --- |
| SQL> @e:/plsql/pragmaexp4.sql;  Enter value for eid: e01  old 6: delete from employee where emp\_id='&eid';  new 6: delete from employee where emp\_id='e01';  cannot be removed as child is present in another table  PL/SQL procedure successfully completed. |

**Exception-Trapping functions**: when an exception occurs in the program the error code and its associated message may not be known. Two functions are used to identify them. They are:

* 1. SQLCODE : the SQLCODE function returns a negative error code number. The number can be assigned to a variable of NUMBER type.
  2. SQLERRM: the SQLERRM function returns the error message associated with the error code. The maximum length of error message is 512 bytes. It can be assigned to a VARCHAR2 data type.

Suppose you have a tabel **company** with the following data:

SQL> desc company;

Name Null? Type

----------------------------------------- -------- ------------------

CID NOT NULL VARCHAR2(7)

CNAME VARCHAR2(15)

CITY VARCHAR2(10)

EID VARCHAR2(7)

SQL> select \* from company;

CID CNAME CITY EID

------- --------------- ---------- -------

c01 tcs chennai 001

c02 tcs chennai 002

c03 cts chennai 003

c04 cts delhi 004

c05 zoho bangalore 005

|  |
| --- |
| declare  v\_cid company.cid%type;  v\_cname company.cname%type:='&cname';  v\_city company.city%type;  v\_eid company.eid%type;  v\_sqlcode number;  v\_msg varchar2(255);  begin  select cid,cname,city,eid into v\_cid,v\_cname,v\_city,v\_eid from company where cname=v\_cname;  dbms\_output.put\_line(v\_cid||' '||v\_cname||' '||v\_city||' '||v\_eid);  exception  when others then  v\_sqlcode:=sqlcode;  v\_msg:=sqlerrm;  dbms\_output.put\_line('error code is '||v\_sqlcode);  dbms\_output.put\_line('error message is '||v\_msg);  end;  / |

output

SQL> @d:/sql/sqlerror.sql

Enter value for cname: tcs

old 3: v\_cname company.cname%type:='&cname';

new 3: v\_cname company.cname%type:='tcs';

error code is -1422

error message is ORA-01422: exact fetch returns more than requested number of

rows

PL/SQL procedure successfully completed.

**User defined exceptions**

When the user come across situations that are specific to a given program, for example checking of user input like birth day input in a specific format, a quantity in an invoice is negative, one can build their own exception.

the three steps to be followed are:

* 1. must declare exception in the declare section.
  2. Must raise the exception in execution section of the program
  3. Must write the handler for the exception

Consider the following table and values in it

SQL> desc emp\_rr;

Name Null? Type

----------------------------------------- -------- ---------------

EID NOT NULL NUMBER(3)

ENAME VARCHAR2(30)

SALARY NUMBER(7,2)

COMMISSION NUMBER(5)

SQL> select \* from emp\_rr;

EID ENAME SALARY COMMISSION

---------- ------------------------------ ---------- ----------

100 vijay 30000 1001

102 anitha 20000 1151

101 lalitha 35000 0

103 krish 25000 -5

104 amala 25000 1200

|  |
| --- |
| declare  invalid\_id EXCEPTION;  invalid\_commission EXCEPTION;  no\_commission EXCEPTION;  v\_comm emp\_rr.commission%type;  v\_ename emp\_rr.ename%type;  empid emp\_rr.eid%type:=&eid;  begin  if empid<0 then  raise invalid\_id;  else  select ename,commission into v\_ename,v\_comm from emp\_rr where eid=empid;  end if;  dbms\_output.put\_line('welcome');  dbms\_output.put\_line(v\_ename||' '||v\_comm);  if v\_comm<0 then  RAISE invalid\_commission;  end if;  if v\_comm=0 then  RAISE no\_commission;  end if;  dbms\_output.put\_line('program over');  EXCEPTION  when invalid\_id then  dbms\_output.put\_line('id must be greater than zero');  when invalid\_commission then  dbms\_output.put\_line('commission negative');  when no\_commission then  dbms\_output.put\_line('no commission value');  when others then  dbms\_output.put\_line('no such id');  end;  / |

Output

|  |
| --- |
| SQL> @e:/plsql/userexp1.sql;  Enter value for eid: 100  old 7: select ename,commission into v\_ename,v\_comm from emp\_rr where eid=&eid;  new 7: select ename,commission into v\_ename,v\_comm from emp\_rr where eid=100;  welcome  vijay 1001  program over  PL/SQL procedure successfully completed.  SQL> /  Enter value for eid: 101  old 7: select ename,commission into v\_ename,v\_comm from emp\_rr where eid=&eid;  new 7: select ename,commission into v\_ename,v\_comm from emp\_rr where eid=101;  welcome  lalitha 0  no commission value  PL/SQL procedure successfully completed.  SQL> /  Enter value for eid: 103  old 7: select ename,commission into v\_ename,v\_comm from emp\_rr where eid=&eid;  new 7: select ename,commission into v\_ename,v\_comm from emp\_rr where eid=103;  welcome  krish -5  commission negative  PL/SQL procedure successfully completed.  SQL> /  Enter value for eid: 0  old 7: empid emp\_rr.eid%type:=&eid;  new 7: empid emp\_rr.eid%type:=0;  no such id  PL/SQL procedure successfully completed. |

RAISE\_APPLICATION\_ERROR

RAISE\_APPLICATION\_ERROR can be used to return user-defined error messages from stored subprograms. The main advantage is, it allows the user to associate own error number and menaningful message with the exception..

Syntax

RAISE\_APPLICATION\_ERROR(*error\_number*, *message*[, {TRUE | FALSE}]);

The ***error\_number*** must fall between -20000 and -20999. This range is reserved by Oracle for programmer use and is never used for predefined Oracle server errors.

***message*** is the user-specified message for the exception. It is character string up to 2,048 bytes long.

TRUE | FALSE is an optional Boolean parameter.

If TRUE, the error is placed on the stack of previous errors

If FALSE—the default—the error replaces all previous errors.

It can be used in two different places:

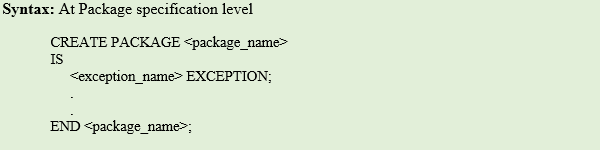
Executable section

Exception section

|  |
| --- |
| declare  v\_eid emp\_rr.eid%type:=&eid;  v\_ename emp\_rr.ename%type;  v\_salary emp\_rr.salary%type;  begin  select ename,salary into v\_ename,v\_salary from emp\_rr where eid=v\_eid;  dbms\_output.put\_line('eid is = '||v\_eid);  dbms\_output.put\_line('ename is = '||v\_ename);  dbms\_output.put\_line('salary is = '||v\_salary);  exception  when no\_data\_found then  RAISE\_APPLICATION\_ERROR(-20201,'this is not a valid employee id');  when too\_many\_rows then  RAISE\_APPLICATION\_ERROR(-20202,'too many employees were found');  end;  / |

Output

|  |
| --- |
| SQL> /  Enter value for eid: 107  old 2: v\_eid emp\_rr.eid%type:=&eid;  new 2: v\_eid emp\_rr.eid%type:=107;  declare  \*  ERROR at line 1:  ORA-20201: this is not a valid employee id  ORA-06512: at line 12 |

[](https://cdn.guru99.com/images/PL-SQL/110215_1024_ExceptionHa3.png)

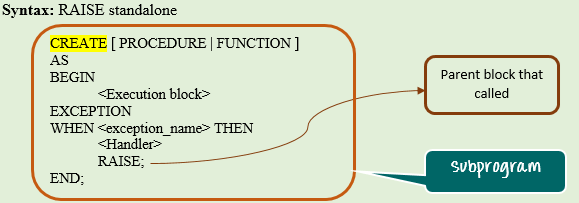
**Syntax Explanation:**

* In the above syntax, the variable 'exception\_name' is defined as 'EXCEPTION' type in the package specification of <package\_name>.
* This can be used in the database wherever package 'package\_name' can be called.

## Raising an Exception for user defined exception

All the predefined exceptions are raised implicitly whenever the error occurs. But the user defined exceptions needs to be raised explicitly. This can be achieved using the keyword 'RAISE'. This can be used in any of the ways mentioned below.

* If 'RAISE' is used separately in the program, then it will propagate the already raised exception to the parent block. Only in exception block can be used as shown below.

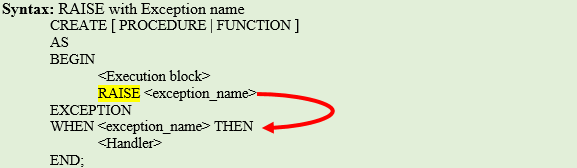
[](https://cdn.guru99.com/images/PL-SQL/110215_1024_ExceptionHa4.png)

**Syntax Explanation:**

* In the above syntax, the keyword RAISE is used in the exception handling block.
* Whenever program encounters exception "exception\_name", the exception is handled and will be completed normally
* But the keyword 'RAISE' in the exception handling part will propagate this particular exception to the parent program.

**Note:** While raising the exception to the parent block the exception that is getting raised should also be visible at parent block, else oracle will throw an error.

* We can use keyword 'RAISE' followed by the exception name to raise that particular user-defined/predefined exception. This can be used in both execution part and in exception handling part to raise the exception.

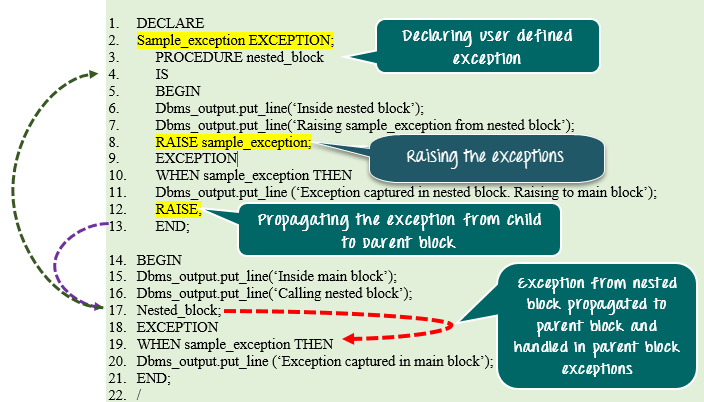
[](https://cdn.guru99.com/images/PL-SQL/110215_1024_ExceptionHa5.png)

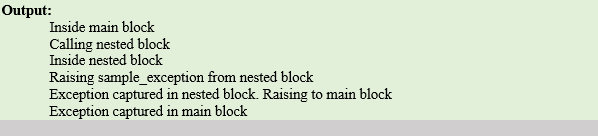
**Syntax Explanation:**

* In the above syntax, the keyword RAISE is used in the execution part followed by exception "exception\_name".
* This will raise this particular exception at the time of execution, and this needs to be handled or raised further.

**Example 1**: In this example, we are going to see

* How to declare the exception
* How to raise the declared exception and
* How to propagate it to the main block

[](https://cdn.guru99.com/images/PL-SQL/110215_1024_ExceptionHa6.png)

[](https://cdn.guru99.com/images/PL-SQL/110215_1024_ExceptionHa7.png)

**Code Explanation:**

* **Code line 2**: Declaring the variable 'sample\_exception' as EXCEPTION type.
* **Code line 3**: Declaring procedure nested\_block.
* **Code line 6**: Printing the statement "Inside nested block".
* **Code line 7:** Printing the statement "Raising sample\_exception from nested block."
* **Code line 8:** Raising the exception using 'RAISE sample\_exception'.
* **Code line 10:** Exception handler for exception sample\_exception in the nested block.
* **Code line 11:** Printing the statement 'Exception captured in nested block. Raising to tmain block'.
* **Code line 12:** Raising the exception to main block (propagating to the main block).
* **Code line 15:** Printing the statement "Inside the main block".
* **Code line 16:** Printing the statement "Calling nested block".
* **Code line 17:** Calling nested\_block procedure.
* **Code line 19:** Exception handler for sample\_exception in the main block.
* **Code line 20:** Printing the statement "Exception captured in the main block."

## Important points to note in Exception

* In function, an exception should always either return value or raise the exception further, else Oracle will throw 'Function returned without a value' error at run-time.
* Transaction control statements can be given at exception handling block.
* SQLERRM and SQLCODE are the in-built functions that will give the exception message and code.
* If an exception is not handled then by default all the active transaction in that session will be rolled back.
* RAISE\_APPLICATION\_ERROR (-<error\_code>, <error\_message>) can be used instead of RAISE to raise the error with user code and message. Error code should be greater than 20000 and prefixed with '-'.

## TCL Statements in PL/SQL

TCL stands for Transaction Control Statements. It will either save the pending transactions or roll back the pending transaction. These statements play the vital role because unless the transaction is saved the changes through DML statements will not be saved in the database. Below are the different TCL statements.

|  |  |
| --- | --- |
| COMMIT | Saves all the pending transaction |
| ROLLBACK | Discard all the pending transaction |
| SAVEPOINT | Creates a point in the transaction till which rollback can be done later |
| ROLLBACK TO | Discard all the pending transaction till the specified <save point> |

The transaction will be complete under the following scenarios.

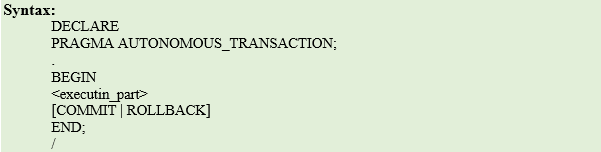
* When any of the above statements is issued (except SAVEPOINT)
* When DDL statements is issued. (DML are auto-commit statements)
* WHEN DCL statements is issued. (DCL are auto-commit statements)

## Autonomous Transaction

In PL/SQL, all the modifications done on data will be termed as a transaction. A transaction is considered as complete when the save/discard is applied to it. If no save/discard is given, then the transaction will not be considered as complete and the modifications done on the data will not be made permanent in the server.

Irrespective of a number of modifications done during a session, PL/SQL will treat the whole modification as a single transaction and saving/discard this transaction affects to the entire pending changes in that session. Autonomous Transaction provides a functionality to the developer in which it allows to do changes in a separate transaction and to save/discard that particular transaction without affecting the main session transaction.

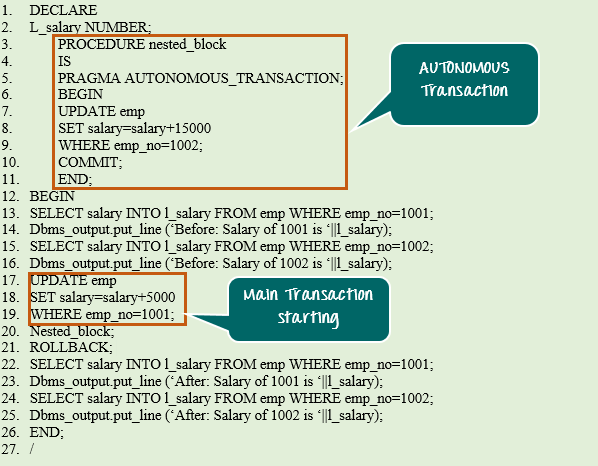
* This autonomous transaction can be specified at subprogram level.
* To make any subprogram to work in a different transaction, the keyword 'PRAGMA AUTONOMOUS\_TRANSATION' should be given in the declarative section of that block.
* It will instruct that compiler to treat this as the separate transaction and saving/discarding inside this block will not reflect in the main transaction.
* Issuing COMMIT or ROLLBACK is mandatory before going out of this autonomous transaction to the main transaction because at any time only one transaction can be active.
* So once we made an autonomous transaction we need to save it and complete the transaction then only we can move back to the main transaction.

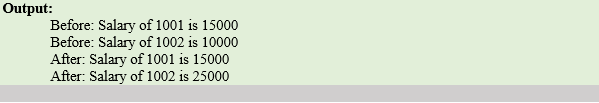
[](https://cdn.guru99.com/images/PL-SQL/110215_1042_SQLinPLSQL16.png)

**Syntax Explanation:**

* In the above syntax, the block has been made as an autonomous transaction.

**Example 1**: In this example, we are going to understand how the autonomous transaction is working.

[](https://cdn.guru99.com/images/PL-SQL/110215_1042_SQLinPLSQL17.png)

[](https://cdn.guru99.com/images/PL-SQL/110215_1042_SQLinPLSQL18.png)

**Code Explanation:**

* **Code line 2**: Declaring l\_salary as NUMBER.
* **Code line 3**: Declaring nested\_block procedure
* **Code line 4**: Making nested\_block procedure as 'AUTONOMOUS\_TRANSACTION'.
* **Code line 7-9:** Increasing the salary for employee number 1002 by 15000.
* **Code line 10:** Committing the transaction.
* **Code line 13-16:** Printing the salary details of employee 1001 and 1002 before changes.
* **Code line 17-19:** Increasing the salary for employee number 1001 by 5000.
* **Code line 20:** Calling the nested\_block procedure;
* **Code line 21:** Discarding the main transaction.
* **Code line 22-25:** Printing the salary details of employee 1001 and 1002 after changes.
* The salary increase for employee number 1001 is not reflected because the main transaction has been discarded. The salary increase for employee number 1002 is reflected because that block has been made as a separate transaction and saved at the end.
* So irrespective of the save/discard at main transaction the changes at autonomous transaction has been saved without affecting the main transaction changes.

## Components of Packages

PL/SQL package basically have two components.

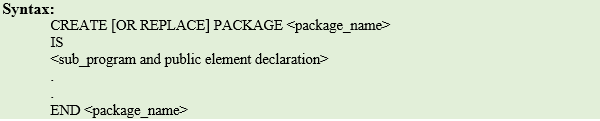
* Package Specification
* Package Body

## Package Specification

Package specification consists of a declaration of all the public variables, cursors, objects, procedures, functions, and exception.

Below are few characteristics of Package specification

* The elements which are all declared in the specification can be accessed from outside of the package. Such element are known as **public** element.
* The package specification is a standalone element that means it can exist alone without package body.
* Whenever a package is referred an instance of the package is created for that particular session.
* After the instance is created for a session, all the package elements that are initiated in that instance are valid till the end of the session.

[](https://cdn.guru99.com/images/PL-SQL/110215_1105_PackagesinP1.png)

**Syntax Explanation:**

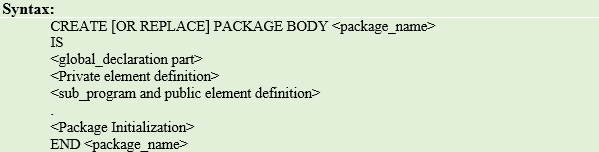
* The above syntax shows the creation of package specification.

## Package Body

It consists of definition of all the elements that are present in the package specification. It can also have definition of elements that are not declared in the specification, these elements are called private elements and can be called only from inside the package.

Below are characteristics of package body.

* It should contain definitions for all the subprograms/cursors that have been declared in the specification.
* It can also have more subprograms or other elements that are not declared in specification. These are called private elements.
* It is a dependable object, and it depends on package specification.
* The state of the package body becomes 'Invalid' every time when the specification is compiled. Therefore, it needs to be recompiled each time after the compilation of specification.
* The private elements should be defined first before they are used in the package body.
* The first part of the package is the global declaration part. This includes variables, cursors and private elements (forward declaration) that is visible to the entire package.
* The last part of the package is Package initialization part that execute one time whenever a package is referred first time in the session.

[](https://cdn.guru99.com/images/PL-SQL/110215_1105_PackagesinP2.png)

**Syntax Explanation:**

* The above syntax shows the creation of package body.

Now we are going to see how to refer package elements in the program.

## Referring Package Elements

Once the elements are declared and defined in the package, we need to refer the elements in order to use them.

All the public elements of the package can be referred by calling the package name followed by the element name separated by period i.e. **'<package\_name>.<element\_name>'.**

The public variable of the package can also be used in the same way to assign and fetch values from them i.e**. '<package\_name>.<variable\_name>'.**

EXAMPLE 1: In the below program a package named ‘**calculate**’ is created. In this prototype declaration of **functions fsum** and **fminus** and **procedure sumn** are declared. The code inside the functions and procedures are not given. Only the prototype declaration is given. This is how you declare a package. We say this as we are giving specification of functions and procedures which we are declaring. In the package body only you write the code for the prototype declarations.

Name of the program is : pack\_sp2.sql

|  |
| --- |
| create or replace package calculate2 as  function fsum(a in number, b in number) return number;  function fminus(a in number, b in number) return number;  procedure sumn(a in number);  end calculate2;  / |

Now create the package body of the above package where the code of respective functions and procedure specification is given above.

The package body will have the same name as package definition. In this example the package has the name **calculate2**and note the package body also has the same.

|  |
| --- |
| create or replace package body calculate2 as  function fsum(a in number, b in number) return number is  c number;  begin  c:=a+b;  return c;  end;  function fminus(a in number, b in number) return number is  c number;  begin  c:=a-b;  return c;  end;  procedure sumn(a in number) is  total number;  begin  total:=0;  for i in 1..a loop  total:=total+i;  end loop;  dbms\_output.put\_line('the sum of '||a||' is = '||total);  end;  end calculate2;  / |

now the functions and procedures defined inside package body can be called with the syntax

<packagename>.<procedure or function name>

The following program the package calculate2.fsum(5,6), calculate2.fminus(9,3) and calculate2.sumn(10) does this work.

|  |
| --- |
| declare  ans number(3);  begin  ans:=calculate2.fsum(5,6);  dbms\_output.put\_line('the sum is '||ans);  ans:=calculate2.fminus(9,3);  dbms\_output.put\_line('the minus is '||ans);  calculate2.sumn(10);  end;  / |

The above programs has to be executed as follows:

**Step 1: the package specification to be executed first as:**

**SQL> @d:\sql\_prgs\pack\_sp2.sql**

**Package created.**

**Step 2: the package body to be executed next**

**SQL> @d:\sql\_prgs\pack\_body2.sql**

**Package body created.**

**Step 3: the program in which the package is called to be executed.**

**SQL> @d:\sql\_prgs\pack\_call2.sql**

**the sum is 11**

**the minus is 6**

**the sum of 10 is = 55**

**PL/SQL procedure successfully completed.**

**Example 2: in this program inside the package called personel , two functions namely get\_empname and get\_salary are defined. This is a package specification.**

|  |
| --- |
| create or replace package **personel** as  function **get\_empname**(neid number) return varchar2;  function **get\_salary**(neid number) return number;  end personel;  / |

In the package body given below the code for the functions specification given in package specification are written. In both package specification and package body the same package name should be given. In this case the name given is **personel**

**File name: emp\_pack\_body.sql**

|  |
| --- |
| create or replace package body **personel** as  function **get\_empname**(neid number) return varchar2 is  v\_empname emp\_rr.ename%type;  begin  select ename into v\_empname from emp\_rr where eid=neid;  return v\_empname;  exception  when no\_data\_found then  dbms\_output.put\_line('no such data');  return null;  when too\_many\_rows then  dbms\_output.put\_line('more than one row returned');  return null;  end; --function ends  function **get\_salary**(neid number) return number is  v\_salary emp\_rr.salary%type;  begin  select salary into v\_salary from emp\_rr where eid=neid;  return v\_salary;  exception  when no\_data\_found then  dbms\_output.put\_line('no such data');  return null;  when too\_many\_rows then  dbms\_output.put\_line('more than one row returned');  return null;  end; --function ends  end personel;  / |

In this program emp\_pack\_call.sql the functions written in package are called by **personel.get\_empname(n\_emp\_id)** and **personel.get\_salary(n\_emp\_id)**

**File name: emp\_pack\_call.sql**

|  |
| --- |
| declare  n\_salary number(7,2);  v\_name varchar2(30);  n\_emp\_id number:=&emp\_id;  begin  v\_name:=personel.get\_empname(n\_emp\_id);  n\_salary:=personel.get\_salary(n\_emp\_id);  if (v\_name is not null and n\_salary is not null) then  dbms\_output.put\_line('Employee: '||v\_name);  dbms\_output.put\_line('Salary : '||n\_salary);  end if;  end;  / |

**Step 1: the package specification to be executed first as:**

**SQL> @d:\sql\_prgs\emp\_pack\_sp1.sql**

**Package created.**

**Step 2: the package body to be executed next**

**SQL> @d:\sql\_prgs\emp\_pack\_body.sql**

**Package body created.**

**Step 3: the program in which the package is called to be executed.**

**SQL> @d:\sql\_prgs\emp\_pack\_call.sql**

|  |
| --- |
| SQL> @e:\plsql\emp\_pack\_call.sql;  Enter value for emp\_id: 100  old 4: n\_emp\_id number:=&emp\_id;  new 4: n\_emp\_id number:=100;  Employee: vijay  Salary : 30000  PL/SQL procedure successfully completed.  SQL> /  Enter value for emp\_id: 104  old 4: n\_emp\_id number:=&emp\_id;  new 4: n\_emp\_id number:=104;  Employee: amala  Salary : 25000  PL/SQL procedure successfully completed.  SQL> @e:/plsql/emp\_pack\_call.sql  Enter value for emp\_id: 107  old 4: n\_emp\_id number:=&emp\_id;  new 4: n\_emp\_id number:=107;  no such data  no such data  PL/SQL procedure successfully completed. |

**Example 3:** Create a package, which consists of two procedures named hire\_employee which will insert new employee details into emp table and another procedure named fire*\_employee* which will delete an employee details from the database.

Table used

|  |
| --- |
| SQL> desc emp\_rr  Name Null? Type  ----------------------------------------- -------- ----------------  EID NOT NULL NUMBER(3)  ENAME VARCHAR2(30)  SALARY NUMBER(7,2)  COMMISSION NUMBER(5) |

**Program for package specification: emp\_pack\_sp2.sql**

|  |
| --- |
| create or replace package hirefire as  procedure hire\_employee(neid emp\_rr.eid%type,vname emp\_rr.ename%type,nsalary emp\_rr.salary%type, ncomm emp\_rr.commission%type);  procedure fire\_employee(neid emp\_rr.eid%type);  end hirefire;  / |

**Program for package body : emp\_pack\_body2.sql**

|  |
| --- |
| create or replace package body hirefire as  procedure hire\_employee(neid emp\_rr.eid%type,vname emp\_rr.ename%type,nsalary emp\_rr.salary%type, ncomm emp\_rr.commission%type)  is  begin  insert into emp\_rr(eid,ename,salary,commission) values(neid,vname,nsalary,ncomm);    end hire\_employee; --procedure hire ends  procedure fire\_employee(neid emp\_rr.eid%type)  is  begin  delete from emp\_rr where eid=neid;  end fire\_employee; --procedure fire ends  end hirefire;  / |

**Program which calls the package : emp\_pac\_call2.sql**

|  |
| --- |
| declare  neid emp\_rr.eid%type;  vname emp\_rr.ename%type;  nsalary emp\_rr.salary%type;  ncomm emp\_rr.commission%type;  begin  hirefire.hire\_employee(&neid,'&vname',&nsalary,&ncomm);    dbms\_output.put\_line('enter the employee id whose data to be deleted');  hirefire.fire\_employee(&neid);  dbms\_output.put\_line('the record of employee id '||neid ||' is deleted');  end;  / |

**Output**

|  |
| --- |
| **SQL> @e:/plsql/emp\_pack\_call2.sql;**  **Enter value for neid: 112**  **Enter value for vname: janani**  **Enter value for nsalary: 30000**  **Enter value for ncomm: 1600**  **old 7: hirefire.hire\_employee(&neid,'&vname',&nsalary,&ncomm);**  **new 7: hirefire.hire\_employee(112,'janani',30000,1600);**  **Enter value for neid: 102**  **old 10: hirefire.fire\_employee(&neid);**  **new 10: hirefire.fire\_employee(102);**  **enter the employee id whose data to be deleted**  **the record of employee id is deleted**  **PL/SQL procedure successfully completed.**  **SQL> select \* from emp\_rr;**  **EID ENAME SALARY COMMISSION**  **---------- ------------------------------ ---------- ----------**  **101 lalitha 35000 0**  **103 krish 25000 -5**  **104 amala 25000 1200**  **110 hari 35000 2000**  **111 rani 20000 0**  **112 janani 30000 1600**  **6 rows selected.** |

## Package Information

Once the package information is created, the package information such as package source, subprogram details and overload details are available in the oracle data definition tables.

Below table gives the data definition table and the package information that is available in the table.

|  |  |  |
| --- | --- | --- |
| **Table Name** | **Description** | **Query** |
| ALL\_OBJECT | Gives the details of the package like object\_id, creation\_date, last\_ddl\_time, etc. It will contain the objects created by all users. | SELECT \* FROM all\_objects where object\_name ='<package\_name>' |
| USER\_OBJECT | Gives the details of the package like object\_id, creation\_date, last\_ddl\_time, etc. It will contain the objects created by the current user. | SELECT \* FROM user\_objects where object\_name ='<package\_name>' |
| ALL\_SOURCE | Gives the source of the objects created by all users. | SELECT \* FROM all\_source where name='<package\_name>' |
| USER\_SOURCE | Gives the source of the objects created by the current user. | SELECT \* FROM user\_source where name='<package\_name>' |
| ALL\_PROCEDURES | Gives the subprogram details like object\_id, overload details, etc created by all users. | SELECT \* FROM all\_procedures Where object\_name='<package\_name>' |
| USER\_PROCEDURES | Gives the subprogram details like object\_id, overload details, etc. created by the current user. | SELECT \* FROM user\_procedures Where object\_name='<package\_name>' |

# PL/SQL Trigger

Trigger is invoked by Oracle engine automatically whenever a specified event occurs.Trigger is stored into database and invoked repeatedly, when specific condition match.

Triggers are stored programs, which are automatically executed or fired when some event occurs.

Triggers are written to be executed in response to any of the following events.

* A database manipulation (DML) statement (DELETE, INSERT, or UPDATE).
* A database definition (DDL) statement (CREATE, ALTER, or DROP).
* A database operation (SERVERERROR, LOGON, LOGOFF, STARTUP, or SHUTDOWN).

Triggers could be defined on the table, view, schema, or database with which the event is associated.

## Advantages of Triggers

These are the following advantages of Triggers:

* Trigger generates some derived column values automatically
* Enforces referential integrity
* Event logging and storing information on table access
* Auditing
* Synchronous replication of tables
* Imposing security authorizations
* Preventing invalid transactions

## Creating a trigger:

**Syntax for creating trigger:**

1. CREATE [OR REPLACE ] TRIGGER trigger\_name
2. {BEFORE | AFTER | INSTEAD OF }
3. {INSERT [OR] | UPDATE [OR] | DELETE}
4. [OF col\_name]
5. ON table\_name
6. [REFERENCING OLD AS o NEW AS n]
7. [FOR EACH ROW]
8. WHEN (condition)
9. DECLARE
10. Declaration-statements
11. BEGIN
12. Executable-statements
13. EXCEPTION
14. Exception-handling-statements
15. END;

**Here,**

* CREATE [OR REPLACE] TRIGGER trigger\_name: It creates or replaces an existing trigger with the trigger\_name.
* {BEFORE | AFTER | INSTEAD OF} : This specifies when the trigger would be executed. The INSTEAD OF clause is used for creating trigger on a view.
* {INSERT [OR] | UPDATE [OR] | DELETE}: This specifies the DML operation.
* [OF col\_name]: This specifies the column name that would be updated.
* [ON table\_name]: This specifies the name of the table associated with the trigger.
* [REFERENCING OLD AS o NEW AS n]: This allows you to refer new and old values for various DML statements, like INSERT, UPDATE, and DELETE.
* [FOR EACH ROW]: This specifies a row level trigger, i.e., the trigger would be executed for each row being affected. Otherwise the trigger will execute just once when the SQL statement is executed, which is called a table level trigger.
* WHEN (condition): This provides a condition for rows for which the trigger would fire. This clause is valid only for row level triggers.

## PL/SQL Trigger Example

|  |
| --- |
| create or replace trigger min\_sal\_chk  before insert or update on works\_csc  for each row  when (new.salary<20000)  begin  raise\_application\_error(-20000, 'sal must be more than 20000');  end;  / |

This trigger checks the value of salary before insert or updates statement and ensures that salary below 20000 is not inserted. It acts before insertion or updation

Execution of triggers during insertion

|  |
| --- |
| SQL> insert into works\_csc values(112,'c1',15000);  insert into works\_csc values(112,'c1',15000)  \*  ERROR at line 1:  ORA-20000: sal must be more than 20000  ORA-06512: at "SYSTEM.MIN\_SAL\_CHK", line 2  ORA-04088: error during execution of trigger 'SYSTEM.MIN\_SAL\_CHK' |

Execution of triggers during updation

|  |
| --- |
| SQL> update works\_csc set salary=15000 where eid=103;  update works\_csc set salary=15000 where eid=103  \*  ERROR at line 1:  ORA-20000: sal must be more than 20000  ORA-06512: at "SYSTEM.MIN\_SAL\_CHK", line 2  ORA-04088: error during execution of trigger 'SYSTEM.MIN\_SAL\_CHK' |

Example 2: this example demonstrates use of trigger to keep information on deleted records.

First create a table to hold deleted records as backup by the following command.

|  |
| --- |
| SQL> create table works\_bkup (eid number(3),cid varchar2(4),salary number(7,2),deldate date);  Table created. |

Now the trigger is created. Whenever a deletion takes place the deleted record is entered into this back up table along with the time of deletion.

|  |
| --- |
| create or replace trigger bkup\_rec  after delete on works\_csc  for each row  begin  insert into works\_bkup values(:old.eid,:old.cid,:old.salary,sysdate);  end;  / |

Execution of trigger.

|  |
| --- |
| SQL> @e:/books/sql\_prgs/works\_trig2.sql;  Trigger created.  SQL> delete from works\_csc where eid=100;  1 row deleted.  SQL> select \* from works\_csc;  EID CID SALARY  ---------- ---- ----------  101 c2 35000  102 c3 35000  103 c4 50000  104 c2 30000  105 c3 30000  106 c1 40000  108 c3 30000  109 c3 28000  8 rows selected.  SQL> select \* from works\_bkup;  EID CID SALARY DELDATE  ---------- ---- ---------- ---------  100 c1 45000 16-SEP-17 |

Example 3: The following trigger execute BEFORE to convert empname field from lowercase to uppercase.

|  |
| --- |
| create or replace trigger emp\_trig  before  insert on employee  for each row  begin  :new.empname:=upper(:new.empname);  end;  / |

Execution of insert command :

|  |
| --- |
| SQL> insert into employee values(113,'rajan','eldams road','chennai');  1 row created. |

The record with eid has empname entered as uppercase.

|  |
| --- |
| SQL> select \* from employee;  EID EMPNAME STREET CITY  ---------- ------------------------------ --------------- ---------------  100 anitha 1st street calcutta  101 aiswarya 2nd street chennai  102 chandra 2nd street chennai  103 hema 3rd street chennai  104 lalitha metha street mumbai  105 raman krishnan street bangalore  106 harini kalam street andhra  107 danush ragav street bangalore  108 david kamaraj street calcutta  109 ananthi rajaji street chennai  113 **RAJAN eldams road chennai**  EID EMPNAME STREET CITY  ---------- ------------------------------ --------------- ---------------  112 krish 3rd street bangalore  12 rows selected. |

Example 4: We write a trigger to ﬁre before the insert takes place.

|  |
| --- |
| SQL> create table person(id int,name varchar2(30),dob date,primary key(id));  Table created. |

On execution of an insert command the trigger will be triggered:

|  |
| --- |
| SQL> insert into person values(10,'anitha','28-sep-1996');  before insert ofanitha  1 row created. |

Example 5: out data base should allow to modify one’s dob. In this case the following trigger helps to achieve this:

|  |
| --- |
| create or replace trigger person\_update\_trig  before  update of dob on person  for each row  begin  raise\_application\_error(-20000,'cannot change date of birth ');  end;  / |

When the updation of dob field takes place the above trigger is triggered.

|  |
| --- |
| SQL> update person set dob='3-aug-1996';  update person set dob='3-aug-1996'  \*  ERROR at line 1:  ORA-20000: cannot change date of birth  ORA-06512: at "SYSTEM.PERSON\_UPDATE\_TRIG", line 2  ORA-04088: error during execution of trigger 'SYSTEM.PERSON\_UPDATE\_TRIG' |

Let's take a simple example to demonstrate the trigger. In this example, we are using the following CUSTOMERS table:

**Create table and have records:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **NAME** | **AGE** | **ADDRESS** | **SALARY** |
| 1 | Ramesh | 23 | Allahabad | 20000 |
| 2 | Suresh | 22 | Kanpur | 22000 |
| 3 | Mahesh | 24 | Ghaziabad | 24000 |
| 4 | Chandan | 25 | Noida | 26000 |
| 5 | Alex | 21 | Paris | 28000 |
| 6 | Sunita | 20 | Delhi | 30000 |

**Create trigger:**

Let's take a program to create a row level trigger for the CUSTOMERS table that would fire for INSERT or UPDATE or DELETE operations performed on the CUSTOMERS table. This trigger will display the salary difference between the old values and new values:

1. CREATE OR REPLACE TRIGGER display\_salary\_changes
2. BEFORE DELETE OR INSERT OR UPDATE ON customers
3. FOR EACH ROW
4. WHEN (NEW.ID > 0)
5. DECLARE
6. sal\_diff number;
7. BEGIN
8. sal\_diff := :NEW.salary  - :OLD.salary;
9. dbms\_output.put\_line('Old salary: ' || :OLD.salary);
10. dbms\_output.put\_line('New salary: ' || :NEW.salary);
11. dbms\_output.put\_line('Salary difference: ' || sal\_diff);
12. END;
13. /

After the execution of the above code at SQL Prompt, it produces the following result.

Trigger created.

**Check the salary difference by procedure:**

Use the following code to get the old salary, new salary and salary difference after the trigger created.

1. DECLARE
2. total\_rows number(2);
3. BEGIN
4. UPDATE  customers
5. SET salary = salary + 5000;
6. IF sql%notfound THEN
7. dbms\_output.put\_line('no customers updated');
8. ELSIF sql%found THEN
9. total\_rows := sql%rowcount;
10. dbms\_output.put\_line( total\_rows || ' customers updated ');
11. END IF;
12. END;
13. /

Output:

Old salary: 20000

New salary: 25000

Salary difference: 5000

Old salary: 22000

New salary: 27000

Salary difference: 5000

Old salary: 24000

New salary: 29000

Salary difference: 5000

Old salary: 26000

New salary: 31000

Salary difference: 5000

Old salary: 28000

New salary: 33000

Salary difference: 5000

Old salary: 30000

New salary: 35000

Salary difference: 5000

6 customers updated

**Note:** As many times you executed this code, the old and new both salary is incremented by 5000 and hence the salary difference is always 5000.

After the execution of above code again, you will get the following result.

Old salary: 25000

New salary: 30000

Salary difference: 5000

Old salary: 27000

New salary: 32000

Salary difference: 5000

Old salary: 29000

New salary: 34000

Salary difference: 5000

Old salary: 31000

New salary: 36000

Salary difference: 5000

Old salary: 33000

New salary: 38000

Salary difference: 5000

Old salary: 35000

New salary: 40000

Salary difference: 5000

6 customers updated

## Important Points

Following are the two very important point and should be noted carefully.

* OLD and NEW references are used for record level triggers these are not avialable for table level triggers.
* If you want to query the table in the same trigger, then you should use the AFTER keyword, because triggers can query the table or change it again only after the initial changes are applied and the table is back in a consistent state.

## Benefits of Triggers

Following are the benefits of triggers.

* Generating some derived column values automatically
* Enforcing referential integrity
* Event logging and storing information on table access
* Auditing
* Synchronous replication of tables
* Imposing security authorizations
* Preventing invalid transactions

## Trigger Classification

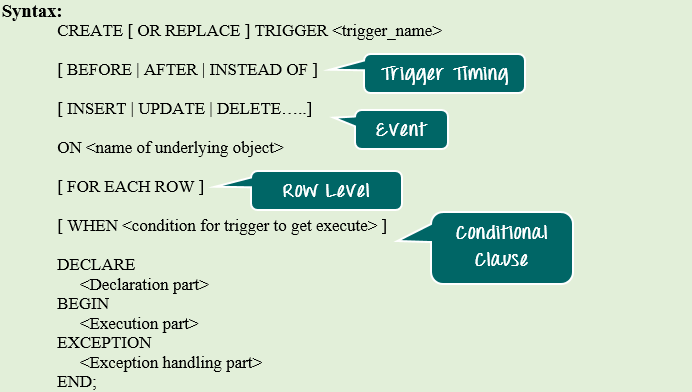
Triggers can be classified based on the following parameters.

* Classification based on the timing
  + BEFORE Trigger: It fires before the specified event has occurred.
  + AFTER Trigger: It fires after the specified event has occurred.
  + INSTEAD OF Trigger: A special type. You will learn more in the further topics. (only for DML )
* Classification based on the level
  + STATEMENT level Trigger: It fires one time for the specified event statement.
  + ROW level Trigger: It fires for each record that got affected in the specified event. (only for DML)
* Classification based on the Event
  + DML Trigger: It fires when the DML event is specified (INSERT/UPDATE/DELETE)
  + DDL Trigger: It fires when the DDL event is specified (CREATE/ALTER)
  + DATABASE Trigger: It fires when the database event is specified (LOGON/LOGOFF/STARTUP/SHUTDOWN)

So each trigger is the combination of above parameters.

## Trigger Creation

Below is the syntax for creating a trigger.

[](https://cdn.guru99.com/images/PL-SQL/110215_1121_TriggersinP1.png)

**Syntax Explanation:**

* The above syntax shows the different optional statements that are present in trigger creation.
* BEFORE/ AFTER will specify the event timings.
* INSERT/UPDATE/LOGON/CREATE/etc. will specify the event for which the trigger needs to be fired.
* ON clause will specify on which object the above mentioned event is valid. For example, this will be the table name on which the DML event may occur in the case of DML Trigger.
* Command "FOR EACH ROW" will specify the ROW level trigger.
* WHEN clause will specify the additional condition in which the trigger needs to fire.
* The declaration part, execution part, exception handling part is same as that of the other PL/SQL blocks. Declaration part and exception handling part are optional.

## :NEW and :OLD Clause

In a row level trigger, the trigger fires for each related row. And sometimes it is required to know the value before and after the DML statement.

Oracle has provided two clauses in the RECORD-level trigger to hold these values. We can use these clauses to refer to the old and new values inside the trigger body.

* :NEW – It holds new value of the columns of the base table/view during the trigger execution
* :OLD – It holds old value of the columns of the base table/view during the trigger execution

These clause should be used based on the DML event. Below table will specify which clause is valid for which DML statement (INSERT/UPDATE/DELETE).

|  |  |  |  |
| --- | --- | --- | --- |
|  | **INSERT** | **UPDATE** | **DELETE** |
| **:NEW** | VALID | VALID | INVALID. There is new value in delete case. |
| **:OLD** | INVALID. There is no old value in insert case | VALID | VALID |

## INSTEAD OF Trigger

"INSTEAD OF trigger" is the special type of trigger. It is used only in DML triggers. It is used when any DML event are going to occur on the complex view.

Consider an example in which a view is made from 3 base tables. When any DML event is issued over this view, that will become invalid because the data is taken from 3 different tables. So in this INSTEAD OF trigger is used. The INSTEAD OF trigger is used to modify the base tables directly instead of modifying the view for the given event.

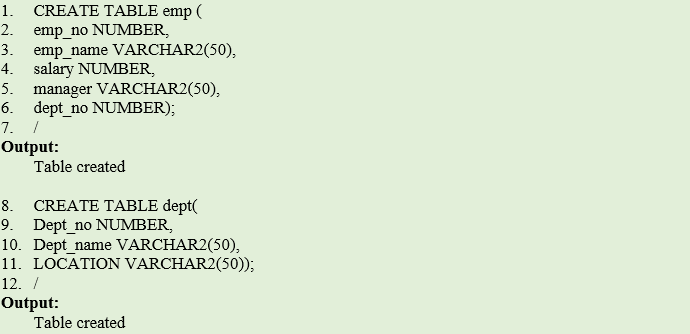
**Example 1**: In this example, we are going to create a complex view from two base table.

* Table\_1 is emp table and
* Table\_2 is department table.

Then we are going to see how the INSTEAD OF trigger is used to issue UPDATE the location detail statement on this complex view. We are also going to see how the :NEW and :OLD is useful in triggers.

* Step 1: Creating table 'emp' and 'dept' with appropriate columns
* Step 2: Populating the table with sample values
* Step 3: Creating view for the above created table
* Step 4: Update of view before instead-of trigger
* Step 5: Creation of instead-of trigger
* Step 6: Update of view after instead-of trigger

**Step 1)** Creating table 'emp' and 'dept' with appropriate columns

[](https://cdn.guru99.com/images/PL-SQL/110215_1121_TriggersinP2.png)

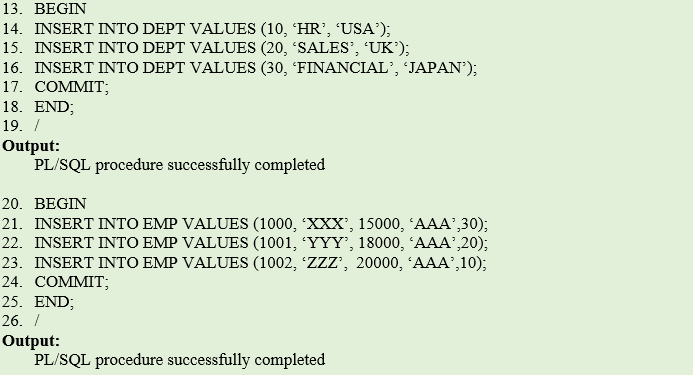
**Code Explanation**

* **Code line 1-7**: Table 'emp' creation.
* **Code line 8-12**: Table 'dept' creation.

**Output**

Table Created

S**tep 2)**Now since we have created the table, we will populate this table with sample values and Creation of Views for the above tables

[](https://cdn.guru99.com/images/PL-SQL/110215_1121_TriggersinP3.png)

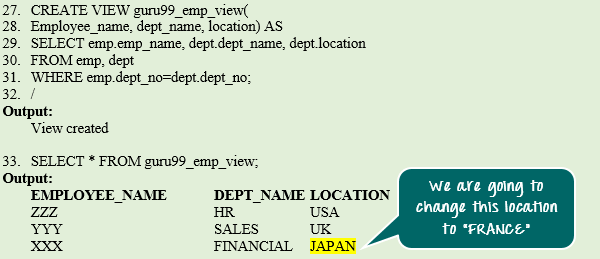
**Code Explanation**

* **Code line 13-19**: Inserting data into 'dept' table.
* **Code line 20-26:** Inserting data into 'emp' table.

**Output**

PL/SQL procedure successfully completed

**Step 3)** Creating a view for the above created table.

[](https://cdn.guru99.com/images/PL-SQL/110215_1121_TriggersinP4.png)

**Code Explanation**

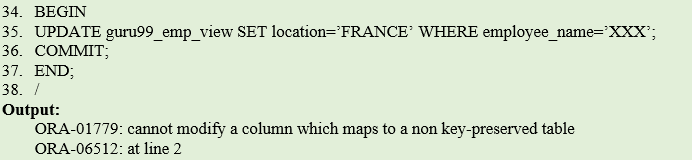
* **Code line 27-32:** Creation of 'guru99\_emp\_view' view.
* **Code line 33:** Querying guru99\_emp\_view.

**Output**

View created

|  |  |  |
| --- | --- | --- |
| **EMPLOYEE\_NAME** | **DEPT\_NAME** | **LOCATION** |
| ZZZ | HR | USA |
| YYY | SALES | UK |
| XXX | FINANCIAL | JAPAN |

**Step 4)** Update of view before instead-of trigger.

[](https://cdn.guru99.com/images/PL-SQL/110215_1121_TriggersinP5.png)

**Code Explanation**

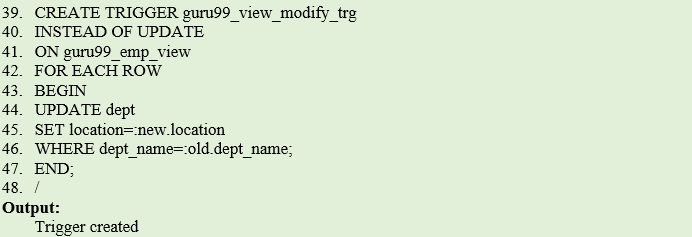
* **Code line 34-38:** Update the location of "XXX" to 'FRANCE'. It raised the exception because the DML statements are not allowed in the complex view.

**Output**

ORA-01779: cannot modify a column which maps to a non key-preserved table

ORA-06512: at line 2

**Step 5)** In order to avoid the error encounter during updating view in the previous step, in this step we are going to use "instead of trigger."

[](https://cdn.guru99.com/images/PL-SQL/110215_1121_TriggersinP6.png)

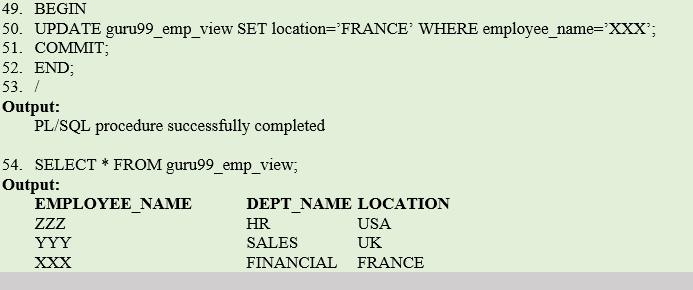
**Code Explanation**

* **Code line 39:** Creation of INSTEAD OF trigger for 'UPDATE' event on the 'guru99\_emp\_view' view at the ROW level. It contains the update statement to update the location in the base table 'dept'.
* **Code line 44:** Update statement uses ':NEW' and ': OLD' to find the value of columns before and after the update.

**Output**

Trigger Created

**Step 6)** Update of view after instead-of trigger. Now the error will not come as the "instead of trigger" will handle the update operation of this complex view. And when the code is executed the location of employee XXX will be updated to "France" from "Japan."

[](https://cdn.guru99.com/images/PL-SQL/110215_1121_TriggersinP7.png)

**Code Explanation:**

* **Code line 49-53:** Update of the location of "XXX" to 'FRANCE'. It is successful because the 'INSTEAD OF' trigger has stopped the actual update statement on view and performed the base table update.
* **Code line 55:** Verifying the updated record.

**Output:**

PL/SQL procedure successfully completed

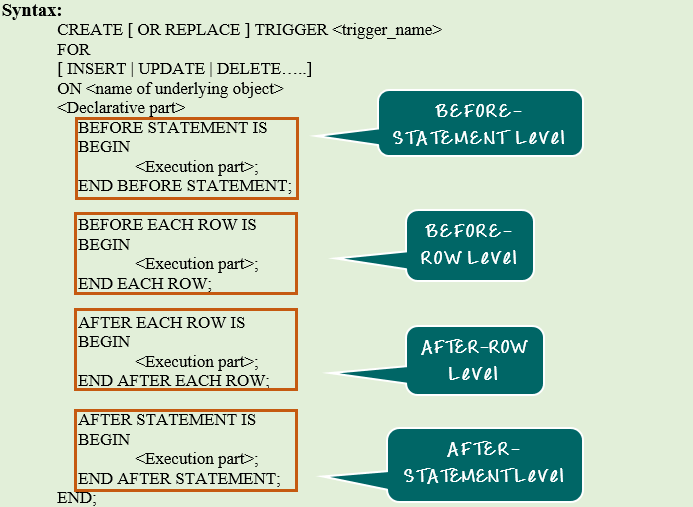
|  |  |  |
| --- | --- | --- |
| **EMPLOYEE\_NAME** | **DEPT\_NAME** | **LOCATION** |
| ZZZ | HR | USA |
| YYY | SALES | UK |
| XXX | FINANCIAL | FRANCE |

## Compound Trigger

Compound trigger is a trigger that allows you to specify actions for each of four timing points in the single trigger body. The 4 different timing point it supports is as below.

* BEFORE STATEMENT – level
* BEFORE ROW – level
* AFTER ROW - level
* AFTER STATEMENT – level

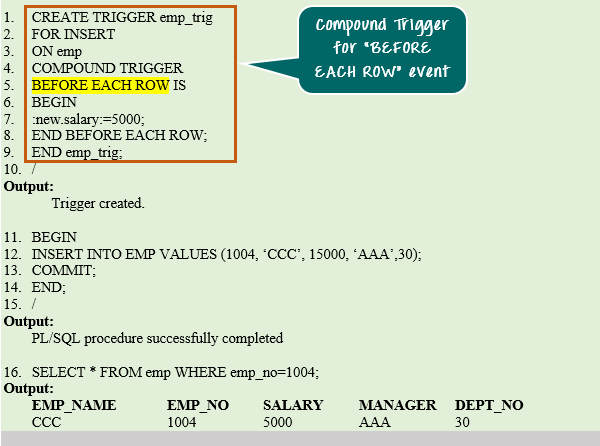
It provides the facility to combine the actions for different timing into the same trigger.

[](https://cdn.guru99.com/images/PL-SQL/110215_1121_TriggersinP8.png)

**Syntax Explanation:**

* The above syntax shows the creation of 'COMPOUND' trigger.
* Declarative section is common for all the execution block in the trigger body.
* These 4 timing blocks can be in any sequence. It is not mandatory to have all these 4 timing blocks. We can create a COMPOUND trigger only for the timings which is required.

**Example 1**: In this example, we are going to create a trigger to auto-populate the salary column with the default value 5000.

[](https://cdn.guru99.com/images/PL-SQL/110215_1121_TriggersinP9.png)

**Code Explanation:**

* **Code line 2-10**: Creation of compound trigger. It is created for timing BEFORE ROW- level to populate the salary with default value 5000. This will change the salary to default value '5000' before inserting the record into the table.
* **Code line 11-14**: Insert the record into 'emp' table.
* **Code line 16**: Verifying the inserted record.

**Output:**

Trigger created

PL/SQL procedure successfully completed

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **EMP\_NAME** | **EMP\_NO** | **SALARY** | **MANAGER** | **DEPT\_NO** |
| CCC | 1004 | 5000 | AAA | 30 |

**Enabling and Disabling Triggers**

Triggers can be enabled or disabled. To enable or disable the trigger, an ALTER (DDL) statement needs to be given for the trigger that disable or enable it.

Below are the syntax for enabling/disabling the triggers.

[Triggers in PL/SQL](https://cdn.guru99.com/images/PL-SQL/110215_1121_TriggersinP10.png)

**Syntax Explanation:**

* The first syntax shows how to enable/disable the single trigger.
* The second statement shows how to enable/disable all the triggers on a particular table.

Varrays

* VARRAYs are of fixed length.
* You specify the length of the array when you define it.
* Arrays of elements of the same type use sequential numbers as a subscript.
* VARRAYS can be used both in PL/SQL and SQL.
* You should use VARRAYs when you know the size of your data set and that size is very stable.
* Varrays are one-dimensional arrays.
* A varray is automatically NULL when it is declared and must be initialized before its elements can be referenced.

Syntax

TYPE TYPE\_NAME IS {VARRAY | VARYING ARRAY} (SIZE\_LIMIT)

OF ELEMENT\_TYPE [NOT NULL]

Example 1: this program stores the string value of digits in varray and display it.

|  |
| --- |
| declare  type digit\_name is varray(10) of varchar2(10) not null;  dn digit\_name;  begin  dn:=digit\_name('zero','one','two','three','four','five','six','seven','eight','nine');  for i in 1..10 loop  dbms\_output.put\_line(i || '->'||dn(i));  end loop;  end;  / |

Output

|  |
| --- |
| SQL> @e:/books/sql\_prgs/varrays\_eg1.sql  1->zero  2->one  3->two  4->three  5->four  6->five  7->six  8->seven  9->eight  10->nine  PL/SQL procedure successfully completed. |

Example:2

|  |
| --- |
| declare  type namearray is varray(5) of varchar2(10);  type grades is varray(5) of integer;  names namearray;  marks grades;  total integer;  begin  names:=namearray('anitha','arthi','aparna','bindhu','banu');  marks:=grades(98,90,91,78,89);  total:=names.count;  dbms\_output.put\_line('Total '|| total ||' students');  for i in 1..total loop  dbms\_output.put\_line('student: '||names(i) ||' marks : '||marks(i));  end loop;  end;  / |

Output:

|  |
| --- |
| SQL> set serveroutput on  SQL> @d:/sql\_prgs/varray\_eg1.sql;  Total 5 students  student: anitha marks : 98  student: arthi marks : 90  student: aparna marks : 91  student: bindhu marks : 78  student: banu marks : 89  PL/SQL procedure successfully completed. |

Example : 3

|  |
| --- |
| declare  type auth\_var is varray(10) of varchar2(30);  x\_auth auth\_var:=auth\_var();  begin  x\_auth.extend;  x\_auth(1):='hello';  x\_auth.extend;  x\_auth(2):='world';  dbms\_output.put\_line(x\_auth(1) || ' & '|| x\_auth(2));  end;  / |

Output

|  |
| --- |
| SQL> @e:/books/sql\_prgs/varrays\_eg4.sql  hello & world  PL/SQL procedure successfully completed. |

In the lines 2 and 3 we define the VARRAY type with a maximum of 10 elements. In line 4 we define the variable x\_auth of auth\_var type and it also initializes the array. A VARRAY can not be used until it is initialized. In line 4 the auth\_var() function actually does the initialization.Once the array is initialized you can extend it and add elements, which is done in lines 6 through 9. We access each element using the VARRAY variable and the index number. When the PL/SQL block ends (or the array variable goes out of scope), the memory used by the array is recovered automatically by the PL/SQL engine. Unlike a cursor, you do not close a collection.

Example 2: this program uses the following collection methods

Extend : will increase the varray

Trim : decreases the size of collection

TRIM removes on element from the end of a collection

TRIM(n) removes n elements from the end of a collection

Count: will return the number of elements in the varray

Limit : checks the maximum size of a collection

First : finds the first subscript in the array

Last: finds the last subscript in the array

Delete : removes all elements from a collection.

|  |
| --- |
| declare  type month\_va is varray(13) of varchar2(20);  v\_month\_va month\_va;  n\_count number;  begin  v\_month\_va:=month\_va('A','B','C','D','E','F','G');  dbms\_output.put\_line('length: ' ||v\_month\_va.count);  v\_month\_va.extend;  v\_month\_va(v\_month\_va.last):='H';  dbms\_output.put\_line('length: ' ||v\_month\_va.count);  for i in v\_month\_va.first..v\_month\_va.last  loop  dbms\_output.put\_line('v\_month(i): ' ||v\_month\_va(i));  end loop;  dbms\_output.put\_line('maximum limit of array is : ' ||v\_month\_va.limit);  dbms\_output.put\_line('first subscript in the array is: ' ||v\_month\_va.first);  dbms\_output.put\_line('last subscript in the array is: ' ||v\_month\_va.last);  dbms\_output.put\_line('value returned by next(3) is: ' ||v\_month\_va.next(3));  v\_month\_va.trim(2);  dbms\_output.put\_line('after trim(2) the number of elements in arrays is: ' ||v\_month\_va.count);  dbms\_output.put\_line('the array elements are: ');  for i in v\_month\_va.first..v\_month\_va.last  loop  dbms\_output.put\_line('v\_month(i): ' ||v\_month\_va(i));  end loop;  end;  / |

Output

|  |
| --- |
| SQL> @e:/books/sql\_prgs/varrays\_eg2.sql  length: 7  length: 8  v\_month(i): A  v\_month(i): B  v\_month(i): C  v\_month(i): D  v\_month(i): E  v\_month(i): F  v\_month(i): G  v\_month(i): H  maximum limit of array is : 13  first subscript in the array is: 1  last subscript in the array is: 8  value returned by next(3) is: 4  after trim(2) the number of elements in arrays is: 6  the array elements are:  v\_month(i): A  v\_month(i): B  v\_month(i): C  v\_month(i): D  v\_month(i): E  v\_month(i): F  PL/SQL procedure successfully completed. |

Example 3: uses collection methods

To demonstrate the use of delete procedure in collections

|  |
| --- |
| declare  type month\_va is varray(13) of varchar2(20);  v\_month\_va month\_va;  n\_count number;  begin  v\_month\_va:=month\_va('A','B','C','D','E','F','G');  dbms\_output.put\_line('length: ' ||v\_month\_va.count);  for i in v\_month\_va.first..v\_month\_va.last  loop  dbms\_output.put\_line('v\_month(i): ' ||v\_month\_va(i));  end loop;  v\_month\_va.delete;  dbms\_output.put\_line('the no. of elements in array is: ' ||v\_month\_va.count);  end;  / |

|  |
| --- |
| SQL> @e:/books/sql\_prgs/varrays\_eg3.sql  length: 7  v\_month(i): A  v\_month(i): B  v\_month(i): C  v\_month(i): D  v\_month(i): E  v\_month(i): F  v\_month(i): G  the no. of elements in array is: 0  PL/SQL procedure successfully completed. |

Elements of a varray could also be a %ROWTYPE of any database table or %TYPE of any database table field. The following example illustrates the concept.

Example :

|  |
| --- |
| --use of varray using cursor  declare  cursor employee\_cur is  select empname from employee;  type empname\_type is varray(20) of employee.empname%type;  name\_va empname\_type :=empname\_type(); --initialize array  v\_counter integer:=0;  begin  for name\_rec in employee\_cur loop  v\_counter:=v\_counter+1;  name\_va.extend;  name\_va(v\_counter):=name\_rec.empname;  dbms\_output.put\_line('employee name: '||name\_va(v\_counter));  end loop;  end;  / |

Output:

|  |
| --- |
| SQL> @e:/books/sql\_prgs/varrays\_eg5.sql  employee name: anitha  employee name: aiswarya  employee name: chandra  employee name: hema  employee name: lalitha  employee name: raman  employee name: harini  employee name: danush  employee name: david  employee name: ananthi  employee name: RAJAN  employee name: krish  PL/SQL procedure successfully completed. |

Tables

A table, like a record is a composite data structure in PL/SQL. It is a single dimensional structure with a collection of elements that store the same type of value.

A PL/SQL table declaration is done in two steps, like a record declaration:

* 1. Declare a PL/SQL table type with a TYPE statement. The structure could use any of the scalare data types.
  2. Declare an actual table based on the type declared in the previous step.

Syntax

TYPE tabletypename IS TABLE OF

Datatype | variablename%TYPE | tablename.columnname%TYPE

[NOT NULL] INDEX BY BINARY\_INTEGER;

For example

TYPE deptname\_table\_type IS TABLE OF dept.Deptname%TYPE

INDEX BY BINARY\_INTEGER;

TYPE major\_table\_type IS TABLE OF VARCHAR2(50)

INDEX BY BINARY\_INTEGER;

Index-By Table

An **index-by** table (also called an **associative array**) is a set of **key-value** pairs. Each key is unique and is used to locate the corresponding value. The key can be either an integer or a string.

An index-by table is created using the following syntax. Here, we are creating an **index-by** table named **table\_name**, the keys of which will be of the subscript\_type and associated values will be of the *element\_type*

TYPE type\_name IS TABLE OF element\_type [NOT NULL] INDEX BY subscript\_type;

table\_name type\_name;