Introduction to Oracle PL/SQL

PL/SQL (Procedural Language/Structured Query Language) is Oracle Corporation's proprietary procedural extension to the SQL database language, used in the Oracle database. Some other SQL database management systems offer similar extensions to the SQL language.

PL/SQL's syntax strongly resembles that of Ada, and just like Ada compilers of the 1980s the PL/SQL runtime system uses Diana as intermediate representation.

The key strength of PL/SQL is its tight integration with the Oracle database.
PL/SQL is one of three languages embedded in the Oracle Database, the other two being SQL and Java.

Oracle's PL/SQL (Procedural Language/SQL) is a database-oriented programming language that is a powerful extension of SQL with procedural capabilities. It is similar to ADA in respect to its syntax, and data types and is used in Oracle databases. Oracle implemented it primarily because of its flexibility and ability to wrap - convert the code to a binary (*.plb) which can't be done with SQL statements. Initially PL/SQL was used on the front end. Lately it obtained more ground on the back end.

All PL/SQL programs are made up of blocks and PL/SQL adds selective and iterative loops (constructs) to SQL.

PL/SQL originates from an older language, Pascal, and carries a lot of similarities to ADA in language structure as much of it is based on ADA. PL/SQL incorporates the Descriptive Intermediate Attribute Notation for Ada (DIANA). DIANA compiles the source interpreted code into an m-code for faster execution. There are many of Ada's concepts in procedures, packages with a package in the most conforming to a similar component of Ada.

Introduction to Oracle 9i:SQL, PLSQL. and SQL *Plus

PL/SQL stands for Procedural Language/SQL. PL/SQL extends SQL by adding constructs found in procedural languages, resulting in a structural language that is more powerful than SQL. The basic unit in PL/SQL is a block. All PL/SQL programs are made up of blocks, which can be nested within each other. Typically, each block performs a logical action in he program. A block has the following structure: DECLARE /* Declarative section: variables, types, and local subprograms. */ BEGIN /* Executable section: procedural and SQL statements go here. */ /* This is the only section of the block that is required. */ EXCEPTION /* Exception handling section: error handling statements go here. */ END; Only the executable section is required. The other sections are optional. The only SQL statements allowed in a PL/SQL program are SELECT,

INSERT, UPDATE, DELETE and several other data manipulation statements plus some transaction control. However, the SELECT statement has a special form in which a single tuple is placed in variables; more on this later. Data definition statements like CREATE, DROP, or ALTER are not allowed. The executable section also contains constructs such as assignments, branches, loops, procedure calls, and triggers, which are all described below (except triggers). PL/SQL is not case sensitive. C style comments (/* ... */) may be used. To execute a PL/SQL program, we must follow the program text itself by * A line with a single dot ("."), and then * A line with run; As with Oracle SQL programs, we can invoke a PL/SQL program either by typing in sqlplus.

PL/SQL can be structured into blocks. It can use conditional statements, loops and branches to control program flow. Scope of the variables can be defined to restrict the visibility within the block.

PL/SQL is a database-orientated programming language. It extends Oracle SQL with procedural capabilities.

PL/SQL programs are organized into functions, procedures and packages and support for object-oriented programming is limited. It's based on the Ada programming language.

The three types of PL/SQL blocks:

1. **Anonymous procedure**

   :It is an unnamed procedure and can't be called. Anonymous procedure is placed where it is to be run, normally attached to a database trigger or application event.

2. **Named procedure**

   :It may be called and it may accept inbound parameters, but named procedure won't explicitly return any value.

3. **Named function**

   :A named function can be called and it may accept inbound parameters. Named function will always return a value.

All of these block types share most PL/SQL features so during this tutorial the features that apply to all block types will be grouped into single subjects.
DECLARE

Definition of any variables or objects that are used within the declared block.
BEGIN
Statements that make up the block.
EXCEPTION
All exception handlers.
END;
End of block marker.

```
DECLARE
TEMP_COST NUMBER(10,2);
BEGIN
SELECT COST INTO TEMP_COST FROM JD11.BOOK WHERE
ISBN = 21;
IF TEMP_COST > 0 THEN
UPDATE JD11.BOOK SET COST = (TEMP_COST*1.175)
WHERE ISBN = 21;
ELSE
UPDATE JD11.BOOK SET COST = 21.32 WHERE ISBN = 21;
END IF;
COMMIT;
Oracle PL-SQL Tutorial
EXCEPTIONOracle PL-SQL Tutorial
Oracle PL-SQL Tutorial
WHEN NO_DATA_FOUND THENOracle PL-SQL Tutorial
Oracle PL-SQL Tutorial
INSERT INTO JD11.ERRORS (CODE, MESSAGE) VALUES(99,
'ISBN 21 NOT FOUND');Oracle PL-SQL Tutorial
Oracle PL-SQL Tutorial
END;
```

As you can see there are several elements in the example that haven't been covered in the SQL tutorial, these elements are the PL/SQL extensions. They include :-

Variables and Constants	These objects are used to store and manipulate block level data. They can be CHAR, VARCHAR2, NUMBER, DATE or BOOLEAN data types.
SQL support	All SQL statements are supported within PL/SQL blocks including transaction control statements.
Composite Datatypes	Records allow groups of fields to be defined and manipulated in PL/SQL blocks.
Flow Control	Ifs, Loops, GOTOs and labels provide conditional actions, tests, branching and iterative program control.
Built In functions	Most SQL data functions are supported within PL/SQL blocks.
Cursor	Cursors (a memory area holding a result set) can be explicitly defined

handling	and manipulated allowing the processing of multiple rows. A group of PL/SQL system attributes provide the ability to test a cursor's internal state.
Exception handling	Blocks have the ability to trap and handle local error conditions (implicit exceptions). You may also self generate explicit exceptions that deal with logic and data errors.
Code storage	Blocks may be stored within an Oracle database as procedures, functions, packages (a group of blocks) and triggers.

Rules of block Structure

1. Every unit of PL/SQL must constitute a block. As a minimum there must be the delimiting words BEGIN and END around the executable statements.
2. SELECT statements within PL/SQL blocks are embedded SQL (an ANSI category). As such they must return one row only. SELECT statements that return no rows or more than one row will generate an error. If you want to deal with groups of rows you must place the returned data into a cursor. The INTO clause is mandatory for SELECT statements within PL/SQL blocks (which are not within a cursor definition), you must store the returned values from a SELECT.
3. If PL/SQL variables or objects are defined for use in a block then you must also have a DECLARE section.
4. If you include an EXCEPTION section the statements within it are only processed if the condition to which they refer occurs. Block execution is terminated after an exception handling routine is executed.
5. PL/SQL blocks may be nested, nesting can occur wherever an executable statement could be placed (including the EXCEPTION section).

Your first example in PL/SQL will be an anonymous block --that is a short program that is ran once, but that is neither named nor stored persistently in the database.

```
SQL> SET SERVEROUTPUT ON
SQL> BEGIN
2 dbms_output.put_line('First World');
3 END;
4 /
```

PL/SQL code is compiled by submitting it to SQL*Plus. Unless your program is an anonymous block, your errors will *not* be reported. Instead, SQL*Plus will display the message ``warning: procedure created with compilation errors". You will then need to type:
SQL> SHOW ERRORS
to see your errors listed. If yo do not understand the error message and you are using Oracle on UNIX, you may be able to get a more detailed description using the `oerr` utility, otherwise use Oracle's documentation (see References section).

SQL> SELECT sysdate FROM DUAL
2 /

```
CREATE OR REPLACE PROCEDURE welcome

IS

user_name VARCHAR2(8) := user;

BEGIN -- `BEGIN' ex

dbms_output.put_line('First World, '

|| user_name || '!');

END;

/
```

Make sure you understand the changes made in the code: Once you have compiled the procedure, execute it using the `EXEC` command.

SQL> EXEC welcome

Generating Output on Screen

The built-in packages offer a number of ways to generate output from within your PL/SQL program. While updating a database table is, of course, a form of "output" from PL/SQL, this chapter shows you how to use two packages that explicitly generate output. UTL_FILE reads and writes information in server-side files, and DBMS_OUTPUT displays information to your screen.
Here is a very good chapter from Orially which helps to understand the complete concept and usage of "**Generating Output on Screen**"

What You Need to Get Started with PL/SQL

Since Oracle is a product designed to be shared, it isn't necessary to have your own private copy of Oracle on your own private machine. You just need an account in an Oracle installation where the administrator will let you experiment with PL/SQL. You can use your desktop machine merely as a tool through which you connect to a database on a different machine. If you don't have that, though, you might have to set up your own

Oracle database.

In the simplest arrangement, you would have the Oracle server running on a machine on your desk, where you would also do all your development. There are four things you will need:

1. Access to a "big enough" machine running an operating system supported by Oracle
2. A licensed copy of Oracle's server software, available free (with some restrictions) from Oracle's web site
3. A text editor
4. An installlation menual

Hardware and Operating System

If you want to install the Enterprise Edition of Oracle9i on a typical Unix machine, Oracle says you need at least the following:

- 256 megabytes of RAM
- 2.5 gigabytes of disk for software and starter database
- 400 megabytes (or more) of swap space during installation

Or, if you want to run the older release, Oracle8i, on a Windows NT or 2000 machine, you'll need a machine something like this:

- Pentium 166MHz or better processor
- 96-megabyte RAM (256 megabytes is recommended)
- 2 gigabytes of disk space

The actual hardware requirements depend on the Oracle version and options you want to use (and, to a lesser extent, on the operating system). As for the operating system, Oracle generally provides licenses for developers (see the next section) on the following:

- Windows NT, Windows 2000, and Windows XP, Professional (some Oracle versions are even available for Windows 98)
- Intel Linux
- Sun Sparc Solaris (a Unix flavor that runs on Sun and Sun-compatible hardware)
- Some Oracle versions are available for other Unix flavors such as Compaq Tru64 Unix and IBM's AIX

It is probably not sufficient to have the version of the operating system that happened to come "out of the box" with the hardware. In addition to matching the exact version number that Oracle supports, you must ensure that the operating system on the machine

has the proper patches (or service packs) installed.

Obtaining the proper version of an operating system for your version of Oracle, and then applying the necessary patches, is usually a task big enough to be annoying. Be sure to follow the instructions in whatever documentation Oracle supplies that is specific to your platform. You should always check the documents that have the name Installation Guide or Release Notes or README in the title. These documents should also contain the exact hardware requirements.

Acquiring Oracle:
Oracle offers a single-user, development-only license for free, as long as you agree to a lot of legal fine print. To obtain a copy of the Oracle server software for use by an individual developer, you can visit the Oracle Technology Network (OTN) web site, sometimes known as Technet, at http://otn.oracle.com. If you have a very fast Internet connection or a lot of time, you can actually download a copy of the software itself. Be warned, though--you may have to download more than a gigabyte of stuff!
If a 48+ hour (at 56K) Internet download isn't your idea of fun, you may be able to order what they call a "CD Pack," currently around $40 in the U.S., or possibly a "Technology Track" subscription, which I bought at one point for about US$200/year. Maybe they have some new deal by now.
When downloading or ordering, you will at some point have to designate which version of which Oracle server you want. After identifying your hardware, you need to choose a version of the database server. My suggestion for beginners is to get the latest available version of the Enterprise Edition unless your organization has a specific requirement for you to learn or support something else. The Personal Edition is probably okay too; I believe it actually includes almost all the features of the Enterprise Edition.

Installing Oracle:
Ihe installer varies according to Oracle version and behaves slightly differently on different platforms. In addition to Oracle's Installation Guide (IG) appropriate to your platform, look in particular for:

- A file in the root directory of the installation media called index.htm or index.html. If you find it, open it in your web browser.
- Anything in a relnotes (Release Notes) subdirectory.
- Anything with README in its filename (and also the lowercase readme, if your operating system is case-sensitive), especially files with rdbms in their names.
- Anything in a doc subdirectory, especially if there is a subdirectory rdbms/doc.

Some of these documents may be available on the OTN web site, but others might only be available after you download and "expand" the software and start poking around in the resulting directories and files. And some may only be available after you've actually installed the software!
If you've never installed Oracle before, I recommend using as many of the default settings as possible. You can almost always rerun the installer later and add or modify the options. I will also mention that if the installer gives you the choice, be sure to install the

built-in web server features, known as "Oracle HTTP Server powered by Apache," or some combination of those words. It may also give a URL to the local web server's administrative page; be sure to write down or copy relevant information.

A Text Editor

A text editor is a program that allows you to create and modify documents such as programs that consist of text only--that is, no fonts, borders, colors, graphics, or other fancy stuff. I've included this requirement as something of a joke, because, as Table 1-3 illustrates, each operating system includes a text editor of some kind.

Environment	Common text editors
DOS	Edit
Windows	Notepad, Wordpad (in text-only mode)
Unix, Linux	vi, GNU emacs
Macintosh	Teachtext, Simpletext

Text editors for various environments

Of course, in addition to the hundreds of different text editors available, there are also commercial programmer's editors and entire interactive development environments available, some of which are built specifically for PL/SQL.

SQL *Plus Commands

SQL*Plus is a command line interface to the Oracle Database. From SQL*Plus, one can issue SQL, PL/SQL and special SQL*Plus commands.

SQL*Plus provides a user-friendly interface which allows you to define and manipulate data in an Oracle database. SQL*Plus adheres to all SQL standards and contains some Oracle-specific add-ons. The "Plus" in SQL*Plus refers to an extension of SQL. SQL*Plus commands allow a user to manipulate and submit SQL statements. Specifically, they enable a user to:

- Enter, edit, store, retrieve, and run SQL statements
- List the column definitions for any table
- Format, perform calculations on, store, and print query results in the form of reports
- Access and copy data between SQL databases

The following is a list of SQL*Plus commands and their functions:

- / - Execute the current SQL statement in the buffer - same as RUN
- ACCEPT - Accept a value from the user and place it into a variable
- APPEND - Add text to the end of the current line of the SQL statement in the buffer
- AUTOTRACE - Trace the execution plan of the SQL statement and gather statistics
- BREAK - Set the formatting behavior for the output of SQL statements
- BTITLE - Place a title on the bottom of each page in the printout from a SQL statement
- CHANGE - Replace text on the current line of the SQL statement with new text
- CLEAR - Clear the buffer
- COLUMN - Change the appearance of an output column from a query
- COMPUTE - Does calculations on rows returned from a SQL statement
- CONNECT - Connect to another Oracle database or to the same Oracle database under a different user name
- COPY - Copy data from one table to another in the same or different databases
- DEL - Delete the current line in the buffer
- DESCRIBE - List the columns with datatypes of a table
- EDIT - Edit the current SQL statement in the buffer using an external editor such as vi or emacs
- EXIT - Exit the SQL*Plus program
- GET - Load a SQL statement into the buffer but do not execute it
- HELP - Obtain help for a SQL*Plus command (In some installations)
- HOST - Drop to the operating system shell
- INPUT - Add one or more lines to the SQL statement in the buffer
- LIST - List the current SQL statement in the buffer
- QUIT - Exit the SQL*Plus program
- REMARK - Place a comment following the REMARK keyword
- RUN - Execute the current SQL statement in the buffer
- SAVE - Save the current SQL statement to a script file
- SET - Set a variable to a new value
- SHOW - Show the current value of a variable
- SPOOL - Send the output from a SQL statement to a file
- START - Load a SQL statement located in a script file and then run that SQL statement
- TIMING - Used to time the execution of SQL statements for performance analysis
- TTITLE - Place a title on the top of each page in the printout from a SQL statement
- UNDEFINE - Delete a user defined variable

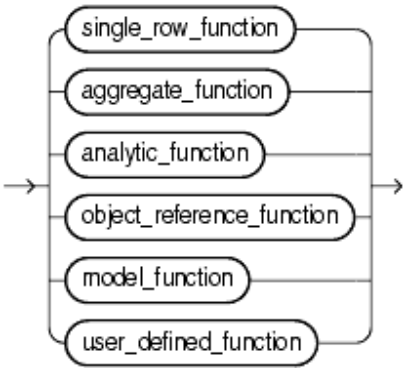
SQL*Plus commands are proprietary to the Oracle SQL*Plus tool. SQL is a standard language that can be used in just about any Relational Database Management System (RDBMS).

SQL Functions

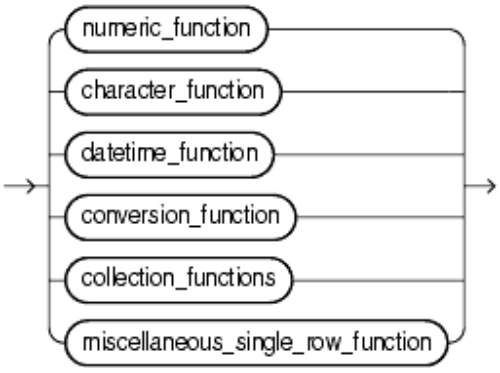
SQL functions are built into **Oracle Database** and are available for use in various appropriate SQL statements. SQL functions are different from user-defined functions written in PL/SQL.

If you call a SQL function with an argument of a datatype other than the datatype expected by the SQL function, then Oracle attempts to convert the argument to the expected datatype before performing the SQL function. If you call a SQL function with a null argument, then the SQL function automatically returns null. The only SQL functions that do not necessarily follow this behavior are CONCAT, NVL, and REPLACE.

In the syntax diagrams for SQL functions, arguments are indicated by their datatypes. When the parameter function appears in SQL syntax, replace it with one of the functions described in this section. Functions are grouped by the datatypes of their arguments and their return values.The syntax showing the categories of functions follows:



Function



Single Row Function

Single-Row Functions

Single-row functions return a single result row for every row of a queried table or view. These functions can appear in select lists, WHERE clauses, START WITH and CONNECT BY clauses, and HAVING clauses.

Numeric Functions

Numeric functions accept numeric input and return numeric values. Most numeric functions that return NUMBER values that are accurate to 38 decimal digits. The transcendental functions COS, COSH, EXP, LN, LOG, SIN, SINH, SQRT, TAN, and TANH are accurate to 36 decimal digits. The transcendental functions ACOS, ASIN, ATAN, and ATAN2 are accurate to 30 decimal digits.

Numeric Functions Returning Character Values

Character functions that return character values return values of the same datatype as the input argument. The length of the value returned by the function is limited by the maximum length of the datatype returned.

- For functions that return CHAR or VARCHAR2, if the length of the return value exceeds the limit, then Oracle Database truncates it and returns the result without an error message.
- For functions that return CLOB values, if the length of the return values exceeds the limit, then Oracle raises an error and returns no data.

Numeric Functions Returning Character Values

Character functions that return number values can take as their argument any character datatype. The character functions that return number values are: ASCII, INSTR, LENGTH, REGEXP_INSTR

Datetime Functions

Datetime functions operate on date (DATE), timestamp (TIMESTAMP, TIMESTAMP WITH TIME ZONE, and TIMESTAMP WITH LOCAL TIME ZONE), and interval (INTERVAL DAY TO SECOND, INTERVAL YEAR TO MONTH) values.
Some of the datetime functions were designed for the Oracle DATE datatype (ADD_MONTHS, CURRENT_DATE, LAST_DAY, NEW_TIME, and NEXT_DAY). If you provide a timestamp value as their argument, Oracle Database internally converts the input type to a DATE value and returns a DATE value. The exceptions are the MONTHS_BETWEEN function, which returns a number, and the ROUND and TRUNC functions, which do not accept timestamp or interval values at all.
The remaining datetime functions were designed to accept any of the three types of data (date, timestamp, and interval) and to return a value of one of these types.

Conversion Functions

Conversion functions convert a value from one datatype to another. Generally, the form of the function names follows the convention datatype TO datatype. The first datatype is the input datatype. The second datatype is the output datatype.

Collection Functions

The collection functions operate on nested tables and varrays.

Miscellaneous Single-Row Functions

These single-row functions do not fall into any of the other single-row function categories.

Aggregate Functions

Aggregate functions return a single result row based on groups of rows, rather than on single rows. Aggregate functions can appear in select lists and in ORDER BY and HAVING clauses. They are commonly used with the GROUP BY clause in a SELECT statement, where Oracle Database divides the rows of a queried table or view into groups. In a query containing a GROUP BY clause, the elements of the select list can be aggregate functions, GROUP BY expressions, constants, or expressions involving one of these. Oracle applies the aggregate functions to each group of rows and returns a single result row for each group.
If you omit the GROUP BY clause, then Oracle applies aggregate functions in the select list to all the rows in the queried table or view. You use aggregate functions in the HAVING clause to eliminate groups from the output based on the results of the aggregate functions, rather than on the values of the individual rows of the queried table or view.
Many (but not all) aggregate functions that take a single argument accept these clauses:

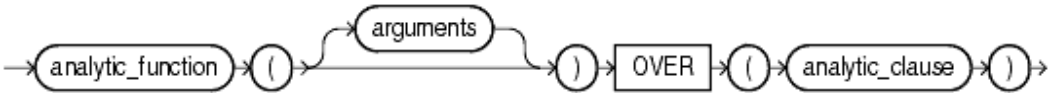
- DISTINCT causes an aggregate function to consider only distinct values of the argument expression.
- ALL causes an aggregate function to consider all values, including all duplicates.

All aggregate functions except COUNT(*) and GROUPING ignore nulls. You can use the NVL function in the argument to an aggregate function to substitute a value for a null. COUNT never returns null, but returns either a number or zero. For all the remaining aggregate functions, if the data set contains no rows, or contains only rows with nulls as arguments to the aggregate function, then the function returns null. You can nest aggregate functions.

Analytic Functions

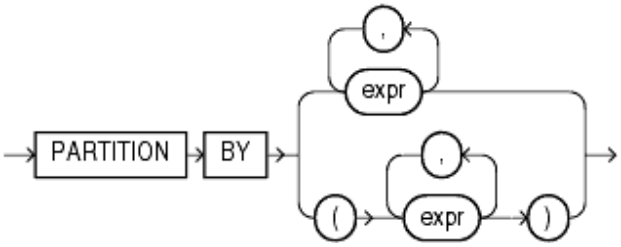
Analytic functions compute an aggregate value based on a group of rows. They differ from aggregate functions in that they return multiple rows for each group. The group of rows is called

a window and is defined by the analytic_clause. For each row, a sliding window of rows is defined. The window determines the range of rows used to perform the calculations for the current row. Window sizes can be based on either a physical number of rows or a logical interval such as time.
Analytic functions are the last set of operations performed in a query except for the final ORDER BY clause. All joins and all WHERE, GROUP BY, and HAVING clauses are completed before the analytic functions are processed. Therefore, analytic functions can appear only in the select list or ORDER BY clause.
Analytic functions are commonly used to compute cumulative, moving, centered, and reporting aggregates.

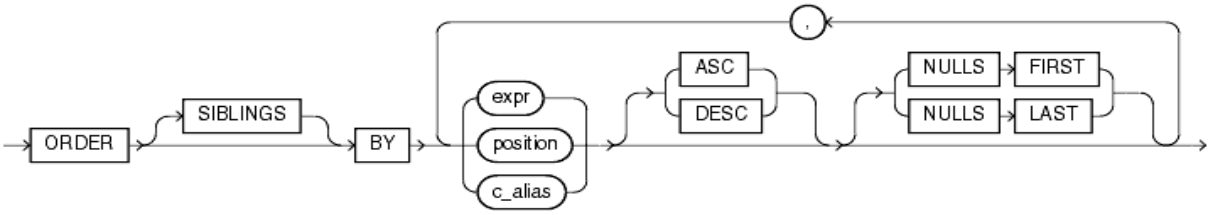


Analytic Function

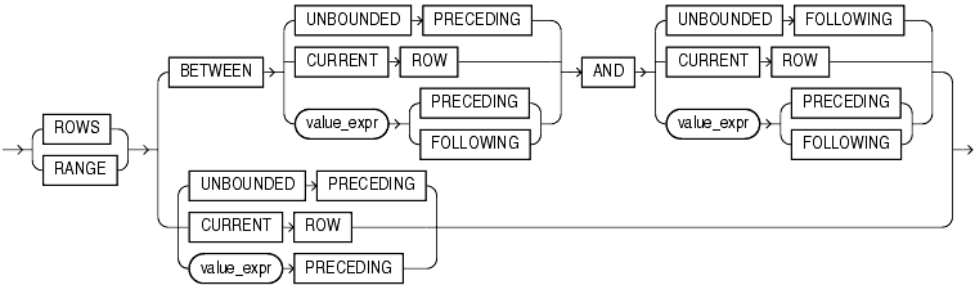
Analytic Clause



Query Partition Clause



Order By Clause



Windowing Clause

Query multiple tables

It's sometimes difficult to know which SQL syntax to use when combining data that spans multiple tables. The following are a few of the more frequently used methods for consolidating queries on multiple tables into a **single** statement.

SELECT

A simple SELECT statement is the most basic way to query multiple tables. More than one table can be called in the FROM clause to combine results from multiple tables. Here's an example of how this works:

SELECT table1.column1, table2.column2 FROM table1, table2 WHERE table1.column1 = table2.column1;

In this example, dot notation (table1.column1) has been used to specify which table the column comes from. If the column in question only appears in one of the referenced tables, the fully qualified name need not be included, but it may be useful to do so for readability.

Tables are separated in the FROM clause by commas. As many tables as needed can be included, although some databases have a limit to what they can efficiently handle.

This syntax is, in effect, a simple INNER JOIN. Some databases treat it exactly the same as an explicit JOIN. The WHERE clause tells the database which fields to correlate, and it returns results as if the tables listed were combined into a single table based on the provided conditions. The conditions for comparison need not necessarily be the same columns as the result set. In the example above, table1.column1 and table2.column1 are used to combine the tables, but table2.column2 is returned.

This functionality can be extended to more than two tables using AND keywords in the WHERE clause.

A combination of tables can be used to restrict the results without actually returning columns from every table. In the example below, table3 is matched up with table1, but nothiong actually has been returned from table3 for display. It merely checks to make sure the relevant column from table1 exists in table3. Also table3 needs to be referenced in the FROM clause for this example.

SELECT table1.column1, table2.column2 FROM table1, table2, table3 WHERE table1.column1 = table2.column1 AND table1.column1 = table3.column1;

This method of querying multiple tables is effectively an implied JOIN. Different databases handle things differently, depending on the optimization engine it uses. Also, neglecting to define the nature of the correlation with a WHERE clause can give undesirable results, such as returning the rogue field in a column associated with every possible result from the rest of the query, as in a CROSS JOIN. A simple SELECT statement is sufficient to combine two or just a few tables.

JOIN

JOIN works in the same way as the SELECT statement above—it returns a result set with columns from different tables. The advantage of using an explicit JOIN over an implied one is greater control over the result set, and possibly improved performance when many tables are involved.

There are several types of JOIN—LEFT, RIGHT, and FULL OUTER; INNER; and CROSS. The type of JOI to be used is determined by the results to be returned. For example, using a LEFT OUTER JOIN will return all relevant rows from the first table listed, while potentially dropping rows from the second table listed if they don't have information that correlates in the first table.

This differs from an INNER JOIN or an implied JOIN. An INNER JOIN will only return rows for which there is data in both tables. The following JOIN statement is equivalent to the first SELECT query above:

```
SELECT table1.column1, table2.column2 FROM table1 INNER JOIN table2 ON
table1.column1 = table2.column1;
```

Subqueries

Subqueries, or subselect statements, are a way to use a result set as a resource in a query. These are often used to limit or refine results rather than run multiple queries or manipulate the data in an application. With a subquery, tables can be referenced to determine inclusion of data or, in some cases, return a column that is the result of a subselect.

The following example uses two tables. One table actually contains the data to be returned, while the other gives a comparison point to determine what data is actually required.

```
SELECT column1 FROM table1 WHERE EXISTS ( SELECT column1 FROM table2
WHERE table1.column1 = table2.column1);
```

One important factor about subqueries is performance. Convenience comes at a price and, depending on the size, number, and complexity of tables and the statements to use, one

may want to allow an application to handle processing. Each query is processed separately in full before being used as a resource for your primary query. If possible, creative use of JOIN statements may provide the same information with less lag time.

UNION

The UNION statement is another way to return information from multiple tables with a single query. The UNION statement allows to perform queries against several tables and return the results in a consolidated set, as in the following example.

SELECT column1, column2, column3 FROM table1 UNION SELECT column1, column2, column3 FROM table2;

This will return a result set with three columns containing data from both queries. By default, the UNION statement will omit duplicates between the tables unless the UNION ALL keyword is used. UNION is helpful when the returned columns from the different tables don't have columns or data that can be compared and joined, or when it prevents running multiple queries and appending the results in your application code.

If the column names don't match while using UNION statement, aliases are used to give results meaningful headers:

SELECT column1, column2 as Two, column3 as Three FROM table1 UNION SELECT column1, column4 as Two, column5 as Three FROM table2;

As with subqueries, UNION statements can create a heavy load on the database server, but for occasional use they can save a lot of time.

When it comes to database queries, there are usually many ways to approach the same problem. These are some of the more frequently used methods for consolidating queries on multiple tables into a single statement. While some of these options may affect performance, practice will help to know when it's appropriate to use each type of query.

Tables and Constraints

A constraint is a rule that the database manager enforces.

There are four types of constraints:

- A **unique constraint** is a rule that forbids duplicate values in one or more columns within a table. Unique and primary keys are the supported unique constraints. For example, a unique constraint can be defined on the supplier identifier in the supplier table to ensure that the same supplier identifier is not

given to two suppliers.
A unique constraint is the rule that the values of a key are valid only if they are unique within a table. Unique constraints are optional and can be defined in the CREATE TABLE or ALTER TABLE statement using the PRIMARY KEY clause or the UNIQUE clause. The columns specified in a unique constraint must be defined as NOT NULL. The database manager uses a unique index to enforce the uniqueness of the key during changes to the columns of the unique constraint. A table can have an arbitrary number of unique constraints, with at most one unique constraint defined as the primary key. A table cannot have more than one unique constraint on the same set of columns.
A unique constraint that is referenced by the foreign key of a referential constraint is called the parent key.
When a unique constraint is defined in a CREATE TABLE statement, a unique index is automatically created by the database manager and designated as a primary or unique system-required index.
When a unique constraint is defined in an ALTER TABLE statement and an index exists on the same columns, that index is designated as unique and system-required. If such an index does not exist, the unique index is automatically created by the database manager and designated as a primary or unique system-required index.
Note that there is a distinction between defining a unique constraint and creating a unique index. Although both enforce uniqueness, a unique index allows nullable columns and generally cannot be used as a parent key.

- A **referential constraint** is a logical rule about values in one or more columns in one or more tables. For example, a set of tables shares information about a corporation's suppliers. Occasionally, a supplier's name changes. You can define a referential constraint stating that the ID of the supplier in a table must match a supplier ID in the supplier information. This constraint prevents insert, update, or delete operations that would otherwise result in missing supplier information.
  Referential integrity is the state of a database in which all values of all foreign keys are valid. A foreign keyis a column or a set of columns in a table whose values are required to match at least one primary key or unique key value of a row in its parent table. A referential constraint is the rule that the values of the foreign key are valid only if one of the following conditions is true:
  They appear as values of a parent key.
  Some component of the foreign key is null.
  The table containing the parent key is called the parent table of the referential constraint, and the table containing the foreign key is said to be a dependent of that table.
  Referential constraints are optional and can be defined in the CREATE TABLE statement or the ALTER TABLE statement. Referential constraints are enforced by the database manager during the execution of INSERT, UPDATE, DELETE, ALTER TABLE, ADD CONSTRAINT, and SET INTEGRITY statements.
  Referential constraints with a delete or an update rule of RESTRICT are enforced before all other referential constraints. Referential constraints with a delete or an update rule of NO ACTION behave like RESTRICT in most cases.

Note that referential constraints, check constraints, and triggers can be combined. Referential integrity rules involve the following concepts and terminology:

- A **table check** constraint sets restrictions on data added to a specific table. For example, a table check constraint can ensure that the salary level for an employee is at least $20,000 whenever salary data is added or updated in a table containing personnel information.
  A table check constraint is a rule that specifies the values allowed in one or more columns of every row in a table. A constraint is optional, and can be defined using the CREATE TABLE or the ALTER TABLE statement. Specifying table check constraints is done through a restricted form of a search condition. One of the restrictions is that a column name in a table check constraint on table T must identify a column of table T.
  A table can have an arbitrary number of table check constraints. A table check constraint is enforced by applying its search condition to each row that is inserted or updated. An error occurs if the result of the search condition is false for any row.
  When one or more table check constraints is defined in the ALTER TABLE statement for a table with existing data, the existing data is checked against the new condition before the ALTER TABLE statement completes. The SET INTEGRITY statement can be used to put the table in check pending state, which allows the ALTER TABLE statement to proceed without checking the data.
- An **informational constraint** is a rule that can be used by the SQL compiler, but that is not enforced by the database manager.
  An informational constraint is a rule that can be used by the SQL compiler to improve the access path to data. Informational constraints are not enforced by the database manager, and are not used for additional verification of data; rather, they are used to improve query performance.
  Use the CREATE TABLE or ALTER TABLE statement to define a referential or table check constraint, specifying constraint attributes that determine whether or not the database manager is to enforce the constraint and whether or not the constraint is to be used for query optimization.

Referential and table check constraints can be turned on or off. It is generally a good idea, for example, to turn off the enforcement of a constraint when large amounts of data are loaded into a database.

Database Objects

Database Objects is a dynamic object-oriented, repository based layer on top of a database. Database, schema, table, column, primary key, and foreign key are a few commonly used database objects. These can be directly created in the Data Definition view.

All objects that belong to the same user are said to be this user's schema. Information about existing objects can be retrieved from dba_objects. It is possible to assign object privileges to objects. These privileges control what other users can do with the objects. In SQL*Plus, information about tables, views, procedures, functions and packages can be shown with describe. Oracle stores relevant information about objects (such as their definitions, used storage and so on) in the data dictionary. The definition of database objects can be retrieved through dbms_metadata.

Following are the examples of a few databse objects:

- Tables
- Views
- Indexes
- Clusters
- Synonyms
- Sequences
- Procedures
- Functions
- Packages
- Triggers

PL/SQL Blocks

A PL/SQL block contains one or more PL/SQL statements. Such a block must at least have the two **keywords** begin and end:


```
begin
  sql statements
end;
```

declare

However, such a block cannot declare any (local) variables. In order to have local variables, the begin...end block must be preceeded with a declare followed by the variable declarations:

```
declare
  variable declarations
begin
```

```
  sql statements
end;
```

exceptions

Additinonally, it is possible to have a exception handler as part of the block. The exception keyword is required for this:

```
declare
  variable declarations
begin
  sql statements
exception
  exception handler
end;
```

```
<<name-for-block>>
declare
  variable declarations
begin
  sql statements
end;
```

A block can be named. Such a label can then be the target of a goto and exit statement.

```
<<name-for-block>>
declare
  variable declarations
begin
  sql statements
end name-for-block;
```

A block can be (recursively) nested, that is, a block can contain other blocks, which in turn can again contain other blocks asf.

```
declare
  variable declarations
begin
  sql statements
declare
  variable declarations
begin
  sql statements
exception
  exception handler
end;
```

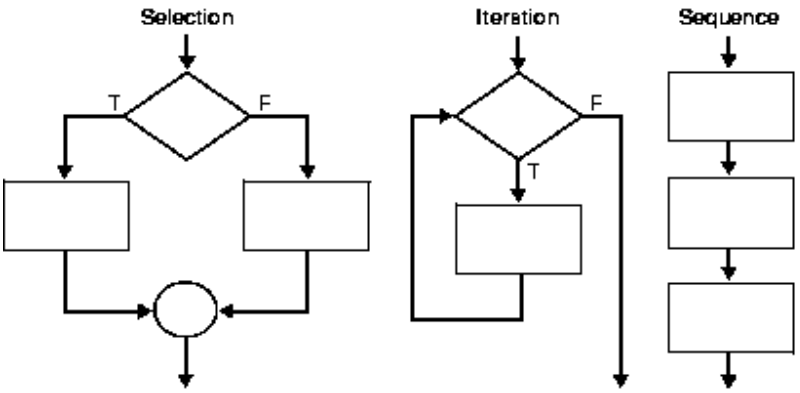
further-sql-statements
exception
  exception handler
end;

PL/SQL Control Structures

In a PL/SQL program statements are connected by simple but powerful control structures that defines flow of control through the program. Control structures that have a **single** entry and exit point and collectively these structures can handle any situation. Their proper use leads naturally to a well-structured program.

Overview of PL/SQL Control Structures

According to the structure theorem, any computer program can be written using the basic control structures shown in the figure below. They can be combined in any way necessary to deal with a given problem.



Control Structures

The selection structure tests a condition, then executes one sequence of statements instead of another, depending on whether the condition is true or false. A condition is any variable or expression that returns a Boolean value (TRUE or FALSE). The iteration structure executes a sequence of statements repeatedly as long as a condition holds true. The sequence structure simply executes a sequence of statements in the order in which they occur.

Conditional Control: IF and CASE Statements

Often, it is necessary to take alternative actions depending on circumstances. The IF statement lets you execute a sequence of statements conditionally. That is, whether the sequence is executed or not depends on the value of a condition. There are three forms of IF statements: IF-THEN, IF-THEN-ELSE, and IF-THEN-ELSIF. The CASE statement is a compact way to evaluate a single condition and choose between many alternative actions.

IF-THEN Statement
The simplest form of IF statement associates a condition with a sequence of statements enclosed by the keywords THEN and END IF (not ENDIF), as follows:

```
IF condition THEN
   sequence_of_statements
END IF;
```

The sequence of statements is executed only if the condition is true. If the condition is false or null, the IF statement does nothing. In either case, control passes to the next statement. An example follows:

```
IF sales > quota THEN
    compute_bonus(empid);
    UPDATE payroll SET pay = pay + bonus WHERE empno = emp_id;
END IF;
```

You might want to place brief IF statements on a single line, as in

```
IF x > y THEN high := x; END IF;
```

IF-THEN-ELSE Statement
The second form of IF statement adds the keyword ELSE followed by an alternative sequence of statements, as follows:

```
IF condition THEN
   sequence_of_statements1
ELSE
   sequence_of_statements2
END IF;
```

The sequence of statements in the ELSE clause is executed only if the condition is false or null. Thus, the ELSE clause ensures that a sequence of statements is executed. In the following example, the first UPDATE statement is executed when the condition is true, but the second UPDATE statement is executed when the condition is false or null:

```
IF trans_type = 'CR' THEN
    UPDATE accounts SET balance = balance + credit WHERE ...
ELSE
    UPDATE accounts SET balance = balance - debit WHERE ...
END IF;
```

The THEN and ELSE clauses can include IF statements. That is, IF statements can be nested, as the following example shows:

```
IF trans_type = 'CR' THEN
   UPDATE accounts SET balance = balance + credit WHERE ...
ELSE
IF new_balance >= minimum_balance THEN
   UPDATE accounts SET balance = balance - debit WHERE ...
ELSE
    RAISE insufficient_funds;
END IF;
END IF;
```

IF-THEN-ELSIF Statement

Sometimes you want to select an action from several mutually exclusive alternatives. The third form of IF statement uses the keyword ELSIF (not ELSEIF) to introduce additional conditions, as follows:

```
IF condition1 THEN
    sequence_of_statements1
ELSIF condition2 THEN
   sequence_of_statements2
ELSE
   sequence_of_statements3
END IF;
```

If the first condition is false or null, the ELSIF clause tests another condition. An IF statement can have any number of ELSIF clauses; the final ELSE clause is optional. Conditions are evaluated one by one from top to bottom. If any condition is true, its associated sequence of statements is executed and control passes to the next statement. If all conditions are false or null, the sequence in the ELSE clause is executed. Consider the following example:

```
BEGIN
  ...
  IF sales > 50000 THEN
    bonus := 1500;
   ELSIF sales > 35000 THEN
     bonus := 500;
  ELSE
     bonus := 100;
   END IF;
  INSERT INTO payroll VALUES (emp_id, bonus, ...);
END;
```

If the value of sales is larger than 50000, the first and second conditions are true. Nevertheless, bonus is assigned the proper value of 1500 because the second condition is never tested. When the first condition is true, its associated statement is executed and control passes to the INSERT statement.

CASE Statement

Like the IF statement, the CASE statement selects one sequence of statements to execute.

However, to select the sequence, the CASE statement uses a selector rather than multiple Boolean expressions. To compare the IF and CASE statements, consider the following code that outputs descriptions of school grades:

```
IF grade = 'A' THEN
   dbms_output.put_line('Excellent');
ELSIF grade = 'B' THEN
   dbms_output.put_line('Very Good');
ELSIF grade = 'C' THEN
   dbms_output.put_line('Good');
ELSIF grade = 'D' THEN
   dbms_output. put_line('Fair');
ELSIF grade = 'F' THEN
   dbms_output.put_line('Poor');
ELSE
   dbms_output.put_line('No such grade');
END IF;
```

Notice the five Boolean expressions. In each instance, we test whether the same variable, grade, is equal to one of five values: 'A', 'B', 'C', 'D', or 'F'. Let us rewrite the preceding code using the CASE statement, as follows:

```
CASE grade
   WHEN 'A' THEN dbms_output.put_line('Excellent');
   WHEN 'B' THEN dbms_output.put_line('Very Good');
   WHEN 'C' THEN dbms_output.put_line('Good');
   WHEN 'D' THEN dbms_output.put_line('Fair');
   WHEN 'F' THEN dbms_output.put_line('Poor');
   ELSE dbms_output.put_line('No such grade');
END CASE;
```

The CASE statement is more readable and more efficient. So, when possible, rewrite lengthy IF-THEN-ELSIF statements as CASE statements.

The CASE statement begins with the keyword CASE. The keyword is followed by a selector, which is the variable grade in the last example. The selector expression can be arbitrarily complex. For example, it can contain function calls. Usually, however, it consists of a single variable. The selector expression is evaluated only once. The value it yields can have any PL/SQL datatype other than BLOB, BFILE, an object type, a PL/SQL record, an index-by-table, a varray, or a nested table.

The selector is followed by one or more WHEN clauses, which are checked sequentially. The value of the selector determines which clause is executed. If the value of the selector equals the value of a WHEN-clause expression, that WHEN clause is executed. For instance, in the last example, if grade equals 'C', the program outputs 'Good'. Execution never falls through; if any WHEN clause is executed, control passes to the next statement.

The ELSE clause works similarly to the ELSE clause in an IF statement. In the last example, if the grade is not one of the choices covered by a WHEN clause, the ELSE clause is selected, and the phrase 'No such grade' is output. The ELSE clause is optional. However, if you omit the ELSE clause, PL/SQL adds the following implicit ELSE

clause:
ELSE RAISE CASE_NOT_FOUND;
If the CASE statement selects the implicit ELSE clause, PL/SQL raises the predefined exception CASE_NOT_FOUND. So, there is always a default action, even when you omit the ELSE clause.
The keywords END CASE terminate the CASE statement. These two keywords must be separated by a space. The CASE statement has the following form:

```
[<,<label_name>>]
CASE selector
   WHEN expression1 THEN sequence_of_statements1;
   WHEN expression2 THEN sequence_of_statements2;
    ...
   WHEN expressionN THEN sequence_of_statementsN;
   [ELSE sequence_of_statementsN+1;]
END CASE [label_name];
```

Like PL/SQL blocks, CASE statements can be labeled. The label, an undeclared identifier enclosed by double angle brackets, must appear at the beginning of the CASE statement. Optionally, the label name can also appear at the end of the CASE statement.
Exceptions raised during the execution of a CASE statement are handled in the usual way. That is, normal execution stops and control transfers to the exception-handling part of your PL/SQL block or subprogram.
An alternative to the CASEstatement is the CASE expression, where each WHEN clause is an expression.

Searched CASE Statement
PL/SQL also provides a searched CASE statement, which has the form:

```
[<,<label_name>>]
CASE
   WHEN search_condition1 THEN sequence_of_statements1;
   WHEN search_condition2 THEN sequence_of_statements2;
   ...
   WHEN search_conditionN THEN sequence_of_statementsN;
   [ELSE sequence_of_statementsN+1;]
END CASE [label_name];
```

The searched CASE statement has no selector. Also, its WHEN clauses contain search conditions that yield a Boolean value, not expressions that can yield a value of any type.
An example follows:

```
CASE
    WHEN grade = 'A' THEN dbms_output.put_line('Excellent');
    WHEN grade = 'B' THEN dbms_output.put_line('Very Good');
   WHEN grade = 'C' THEN dbms_output.put_line('Good');
   WHEN grade = 'D' THEN dbms_output.put_line('Fair');
    WHEN grade = 'F' THEN dbms_output.put_line('Poor');
   ELSE dbms_output.put_line('No such grade');
END CASE;
```

The search conditions are evaluated sequentially. The Boolean value of each search condition determines which WHEN clause is executed. If a search condition yields TRUE, its WHEN clause is executed. If any WHEN clause is executed, control passes to the next statement, so subsequent search conditions are not evaluated.
If none of the search conditions yields TRUE, the ELSE clause is executed. The ELSE clause is optional. However, if you omit the ELSE clause, PL/SQL adds the following implicit ELSE clause:
ELSE RAISE CASE_NOT_FOUND;
Exceptions raised during the execution of a searched CASE statement are handled in the usual way. That is, normal execution stops and control transfers to the exception-handling part of your PL/SQL block or subprogram.

Guidelines for PL/SQL Conditional Statements
Avoid clumsy IF statements like those in the following example:

```
IF new_balance < minimum_balance THEN
    overdrawn := TRUE;
ELSE
    overdrawn := FALSE;
END IF;
... IF overdrawn = TRUE THEN
    RAISE insufficient_funds;
END IF;
```

This code disregards two useful facts. First, the value of a Boolean expression can be assigned directly to a Boolean variable. So, you can replace the first IF statement with a simple assignment, as follows:
overdrawn := new_balance < minimum_balance;
Second, a Boolean variable is itself either true or false. So, you can simplify the condition in the second IF statement, as follows:
IF overdrawn THEN ...
When possible, use the ELSIF clause instead of nested IF statements. That way, your code will be easier to read and understand. Compare the following IF statements:

```
IF condition1 THEN
    statement1;
ELSE
   IF condition2 THEN
      statement2;
   ELSE
      IF condition3 THEN
         statement3;
      END IF;
   END IF;
END IF;
```

```
IF condition1 THEN
   statement1;
```

```
ELSIF condition2 THEN
   statement2;
ELSIF condition3 THEN
   statement3;
END IF;
   END IF;
END IF;
```

These statements are logically equivalent, but the first statement obscures the flow of logic, whereas the second statement reveals it.

If you are comparing a single expression to multiple values, you can simplify the logic by using a single CASE statement instead of an IF with several ELSIF clauses.

Iterative Control: LOOP and EXIT Statements

LOOP statements let you execute a sequence of statements multiple times. There are three forms of LOOP statements: LOOP, WHILE-LOOP, and FOR-LOOP

LOOP

The simplest form of LOOP statement is the basic (or infinite) loop, which encloses a sequence of statements between the keywords LOOP and END LOOP, as follows:

```
LOOP
    sequence_of_statements
END LOOP;
```

With each iteration of the loop, the sequence of statements is executed, then control resumes at the top of the loop. If further processing is undesirable or impossible, you can use an EXIT statement to complete the loop. You can place one or more EXIT statements anywhere inside a loop, but nowhere outside a loop. There are two forms of EXIT statements: EXIT and EXIT-WHEN.

EXIT

The EXIT statement forces a loop to complete unconditionally. When an EXIT statement is encountered, the loop completes immediately and control passes to the next statement. An example follows:

```
LOOP
   ...
   IF credit_rating < 3 THEN
      ...
       EXIT; -- exit loop immediately
   END IF;
END LOOP;
-- control resumes here
```

The next example shows that you cannot use the EXIT statement to complete a PL/SQL block:

```
BEGIN
   ...     IF credit_rating < 3 THEN
```

```
      ...
      EXIT; -- not allowed
   END IF;
END;
```
Remember, the EXIT statement must be placed inside a loop. To complete a PL/SQL block before its normal end is reached, you can use the RETURN statement.

EXIT-WHEN

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

```
LOOP
    FETCH c1 INTO ...
   EXIT WHEN c1%NOTFOUND; -- exit loop if condition is true
...
   END LOOP;
CLOSE c1;
-- control resumes here
```
Until the condition is true, the loop cannot complete. So, a statement inside the loop must change the value of the condition. In the last example, if the FETCH statement returns a row, the condition is false. When the FETCH statement fails to return a row, the condition is true, the loop completes, and control passes to the CLOSE statement.
```
IF count > 100 THEN
   EXIT;
END IF;

EXIT WHEN count > 100;
```
These statements are logically equivalent, but the EXIT-WHEN statement is easier to read and understand.

FOR-LOOP

Whereas the number of iterations through a WHILE loop is unknown until the loop completes, the number of iterations through a FOR loop is known before the loop is entered. FOR loops iterate over a specified range of integers. The range is part of an iteration scheme, which is enclosed by the keywords FOR and LOOP. A double dot (..) serves as the range operator. The syntax follows:

```
FOR counter IN [REVERSE] lower_bound..higher_bound LOOP
   sequence_of_statements
END LOOP;
```
The range is evaluated when the FOR loop is first entered and is never re-evaluated. As the next example shows, the sequence of statements is executed once for each integer in the range. After each iteration, the loop counter is incremented.
```
FOR i IN 1..3 LOOP -- assign the values 1,2,3 to i
```

```
   sequence_of_statements -- executes three times
END LOOP;
```
The following example shows that if the lower bound equals the higher bound, the sequence of statements is executed once:
```
FOR i IN 3..3 LOOP -- assign the value 3 to i
   sequence_of_statements -- executes one time
END LOOP;
```
By default, iteration proceeds upward from the lower bound to the higher bound. However, as the example below shows, if you use the keyword REVERSE, iteration proceeds downward from the higher bound to the lower bound. After each iteration, the loop counter is decremented. Nevertheless, you write the range bounds in ascending (not descending) order.
```
FOR i IN REVERSE 1..3 LOOP -- assign the values 3,2,1 to i
    sequence_of_statements -- executes three times
END LOOP;
```
Inside a FOR loop, the loop counter can be referenced like a constant but cannot be assigned values, as the following example shows:
```
FOR ctr IN 1..10 LOOP      IF NOT finished THEN          INSERT INTO ... VALUES
(ctr, ...); -- legal          factor := ctr * 2; -- legal     ELSE          ctr := 10; -- not
allowed      END IF; END LOOP;
```

WHILE-LOOP

The WHILE-LOOP statement associates a condition with a sequence of statements enclosed by the keywords LOOP and END LOOP, as follows:

```
WHILE condition LOOP
   sequence_of_statements
END LOOP;
```
Before each iteration of the loop, the condition is evaluated. If the condition is true, the sequence of statements is executed, then control resumes at the top of the loop. If the condition is false or null, the loop is bypassed and control passes to the next statement. An example follows:
```
WHILE total <= 25000 LOOP
   ...
   SELECT sal INTO salary FROM emp WHERE ...
   total := total + salary;
END LOOP;
```
The number of iterations depends on the condition and is unknown until the loop completes. The condition is tested at the top of the loop, so the sequence might execute zero times. In the last example, if the initial value of total is larger than 25000, the condition is false and the loop is bypassed. Some languages have a LOOP UNTIL or REPEAT UNTIL structure, which tests the condition at the bottom of the loop instead of at the top. Therefore, the sequence of statements is executed at least once. PL/SQL has no such structure, but you can easily build one, as follows:
```
LOOP
   sequence_of_statements
```

```
   EXIT WHEN boolean_expression;
END LOOP;
```
To ensure that a WHILE loop executes at least once, use an initialized Boolean variable in the condition, as follows:
```
done := FALSE;
WHILE NOT done LOOP
   sequence_of_statements
   done := boolean_expression
END LOOP;
```
A statement inside the loop must assign a new value to the Boolean variable. Otherwise, you have an infinite loop. For example, the following LOOP statements are logically equivalent:
```
LOOP
   ...
END LOOP;

WHILE TRUE LOOP
   ...
END LOOP;
```

Sequential Control: GOTO and NULL Statements

Unlike the IF and LOOP statements, the GOTO and NULL statements are not crucial to PL/SQL programming. The structure of PL/SQL is such that the GOTO statement is seldom needed. Occasionally, it can simplify logic enough to warrant its use. The NULL statement can improve readability by making the meaning and action of conditional statements clear.
Overuse of GOTO statements can result in complex, unstructured code (sometimes called spaghetti code) that is hard to understand and maintain. So, use GOTO statements sparingly. For example, to branch from a deeply nested structure to an error-handling routine, raise an exception rather than use a GOTO statement.

NULL Statement
The NULL statement does nothing other than pass control to the next statement. In a conditional construct, the NULL statement tells readers that a possibility has been considered, but no action is necessary. In the following example, the NULL statement shows that no action is taken for unnamed exceptions:

```
EXCEPTION
   WHEN ZERO_DIVIDE THEN        ROLLBACK;     WHEN VALUE_ERROR
THEN        INSERT INTO errors VALUES ...        COMMIT;     WHEN OTHERS
THEN        NULL; END;
```
In IF statements or other places that require at least one executable statement, the NULL statement to satisfy the syntax. In the following example, the NULL statement emphasizes that only top-rated employees get bonuses:
```
IF rating > 90 THEN
```

```
   compute_bonus(emp_id);
ELSE
   NULL;
END IF;
```
Also, the NULL statement is a handy way to create stubs when designing applications from the top down. A stub is dummy subprogram that lets you defer the definition of a procedure or function until you test and debug the main program. In the following example, the NULL statement meets the requirement that at least one statement must appear in the executable part of a subprogram:
```
PROCEDURE debit_account (acct_id INTEGER, amount REAL) IS
BEGIN
   NULL;
END debit_account;
```

GOTO Statement

The GOTO statement branches to a label unconditionally. The label must be unique within its scope and must precede an executable statement or a PL/SQL block. When executed, the GOTO statement transfers control to the labeled statement or block. In the following example, you go to an executable statement farther down in a sequence of statements:

```
BEGIN
    ...
   GOTO insert_row;
   ...     <<insert_row>>
   INSERT INTO emp VALUES ...
END;
```
In the next example, you go to a PL/SQL block farther up in a sequence of statements:
```
BEGIN
   ...
   <<update_row>>
   BEGIN
      UPDATE emp SET ...
      ...
   END;
   ...
   GOTO update_row;
   ...
END;
```
The label end_loop is not allowed if it does not precede an executable statement, just add the NULL statement to debug the statement before end_loop as NULL is executable. The GOTO statement branches to the first enclosing block in which the referenced label appears.

Oracle PL/SQL Cursors

The information within a **SQL Statement** can be manipulated by means of assigning a name to a "select" statement, this concept is known as cursor. A cursor is used for processing individual rows returned as a result of a query.

Explicit Cursors
Explicit cursors are SELECT statements that are DECLAREd explicitly in the declaration section of the current block or in a package specification. Use OPEN, FETCH, and CLOSE in the execution or exception sections of your programs.

Implicit Cursors
Whenever a SQL statement is directly in the execution or exception section of a PL/SQL block, you are working with implicit cursors. These statements include INSERT, UPDATE, DELETE, and SELECT INTO statements. Unlike explicit cursors, implicit cursors do not need to be declared, OPENed, FETCHed, or CLOSEd.

Exception Handling in PL/SQL

PL/SQL provides a feature to handle the Exceptions which occur in a PL/SQL Block known as **exception Handling**

. Using Exception Handling we can test the code and avoid it from exiting abruptly. When an exception occurs a messages which explains its cause is recieved.

PL/SQL Exception message consists of three parts.

1. Type of Exception
2. An Error Code
3. A message

By Handling the exceptions we can ensure a PL/SQL block does not exit abruptly.

The General Syntax for coding the exception section.

```
DECLARE
  Declaration section
BEGIN
```

```
   Exception section
EXCEPTION
WHEN ex_name1 THEN
   -Error handling statements
WHEN ex_name2 THEN
  -Error handling statements
WHEN Others THEN
  -Error handling statements
END;
```

General PL/SQL statments can be used in the Exception Block.

When an exception is raised, Oracle searches for an appropriate exception handler in the exception section. For example in the above example, if the error raised is 'ex_name1 ', then the error is handled according to the statements under it. Since, it is not possible to determine all the possible runtime errors during testing fo the code, the 'WHEN Others' exception is used to manage the exceptions that are not explicitly handled. Only one exception can be raised in a Block and the control does not return to the Execution Section after the error is handled.
If there are nested PL/SQL blocks like this.

```
DELCARE
  Declaration section BEGIN
  DECLARE
   Declaration section
BEGIN
  Execution section
EXCEPTION
 Exception section
END;
EXCEPTION
 Exception section
END;
```

In the above case, if the exception is raised in the inner block it should be handled in the exception block of the inner PL/SQL block else the control moves to the Exception block of the next upper PL/SQL Block. If none of the blocks handle the exception the program ends abruptly with an error.

Types of Exception.

There are 3 types of Exceptions.

- **Named System Exceptions :**System exceptions are automatically raised by Oracle, when a program violates a RDBMS rule. There are some system exceptions which are raised frequently, so they are pre-defined and given a name

in Oracle which are known as Named System Exceptions.

- **Unnamed System Exceptions :**Those system exception for which oracle does not provide a name is known as unamed system exception. These exception do not occur frequently. These Exceptions have a code and an associated message.
  There are two ways to handle unnamed sysyem exceptions:
  1. By using the WHEN OTHERS exception handler, or
  2. By associating the exception code to a name and using it as a named exception.
  We can assign a name to unnamed system exceptions using a Pragma called EXCEPTION_INIT. EXCEPTION_INIT will associate a predefined Oracle error number to a programmer_defined exception name.
  Steps to be followed to use unnamed system exceptions are:
  • They are raised implicitly.
  • If they are not handled in WHEN Others they must be handled explicity.
  • To handle the exception explicity, they must be declared using Pragma EXCEPTION_INIT as given above and handled referecing the user-defined exception name in the exception section.
- **User-defined Exceptions :**Apart from sytem exceptions we can explicity define exceptions based on business rules. These are known as user-defined exceptions.
  Steps to be followed to use user-defined exceptions:
  • They should be explicitly declared in the declaration section.
  • They should be explicitly raised in the Execution Section.
  • They should be handled by referencing the user-defined exception name in the exception section.

RAISE_APPLICATION_ERROR ( )

RAISE_APPLICATION_ERROR is a built-in procedure in oracle which is used to display the user-defined error messages along with the error number whose range is in between -20000 and -20999. Whenever a message is displayed using RAISE_APPLICATION_ERROR, all previous transactions which are not committed within the PL/SQL Block are rolled back automatically (i.e. change due to INSERT, UPDATE, or DELETE statements).

RAISE_APPLICATION_ERROR raises an exception but does not handle it.
RAISE_APPLICATION_ERROR is used for the following reasons:

- to create a unique id for an user-defined exception.
- to make the user-defined exception look like an Oracle error.

PL/SQL Collections

In PL/SQL a collection variable is a variable that can store zero, one ore more elements

of a specific type (either an internal data type or a user defined data types). The type of the variable is itself a user defined type. Since a collection is a user defined type, a collection type can store collections as well.

Just about all modern programming languages provide support for collections. A collection can be loosely defined as a group of ordered elements, all of the same type, that allows programmatic access to its elements through an index. Commonly used collection types used in the programming world include arrays, maps, and lists.

Different collection types

There are three collection types in PL/SQL:

- index-by tables, also known as associative arrays
- varrays
- nested tables

Nested tables extend the functionality of index-by tables. The main difference is that nested tables can be stored in a table column while index by tables can not.

Use for collections

This example shows how a table can be returned from a pl/sql function.
poor man's text index is an example that uses collections to search in the middle of words in a table. (ie where filed like '%word%')
Index by tables are also used in bulk collect statements.

Collection functions

The collection functions operate on nested tables and varrays. cardinality
collect
powermultiset
powermultiset_by_cardinality
set

Procedures and Functions

PL/SQL functions returns a scalar value and PL/SQL procedures return nothing. Both can take zero or more number of parameters as input or output. The special feature about

PL/SQL is that a procedure/function argument can be of input (indicating the argument is read-only), output (indicating the argument is write-only) or both (both readable and writable).

Functions in PL/SQL are a collection of SQL and PL/SQL statements that perform a task and should return a value to the calling environment.

```
CREATE OR REPLACE FUNCTION <<function_name>> [(input/output variable
declarations)] RETURN
return_type <<IS|AS>>
   [declaration block]
BEGIN
<<PL/SQL block WITH RETURN statement>>
[EXCEPTION
   EXCEPTION block]
END;
```

Procedures are the same as Functions, in that they are also used to perform some task with the difference being that procedures cannot be used in a SQL statement and although they can have multiple out parameters they do not return a value.

Procedures are traditionally the workhorse of the coding world and functions are traditionally the smaller, more specific pieces of code. In general, if you need to update the chart of accounts, you would write a procedure. If you need to retrieve the organization code for a particular GL account, you would write a function.
Here are a few more differences between a procedure and a function:

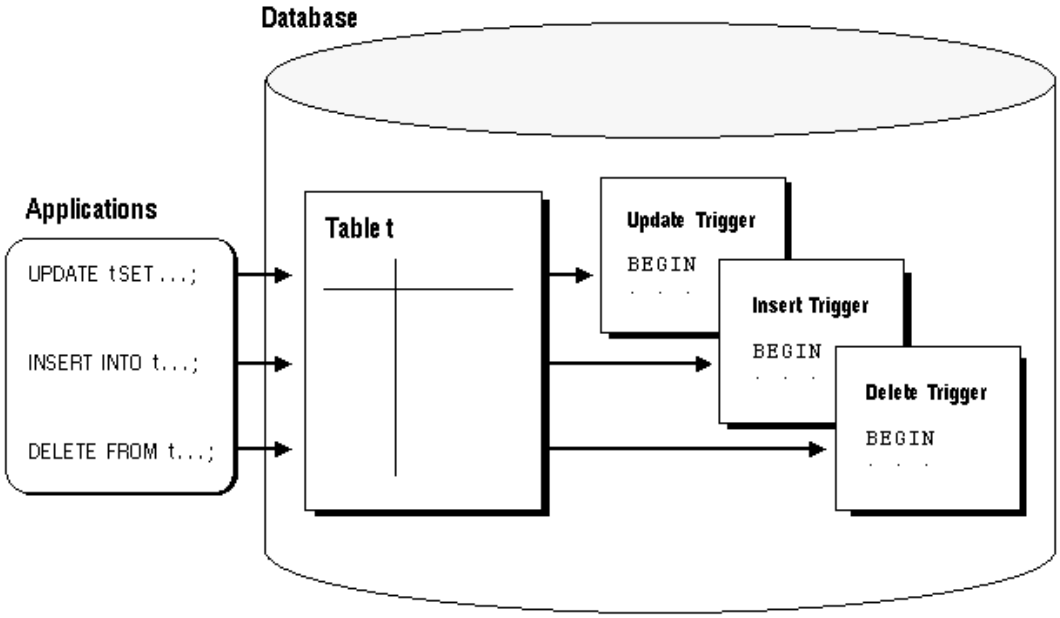
- A function MUST return a value
- A procedure cannot return a value
- Procedures and functions can both return data in OUT and IN OUT parameters
- The return statement in a function returns control to the calling program and returns the results of the function
- The return statement of a procedure returns control to the calling program and cannot return a value
- Functions can be called from SQL, procedure cannot
- Functions are considered expressions, procedure are not

Oracle PL/SQL Cursors

Trigger are procedures that are stored in the database and implicitly executed ("fired") when a table is

modified. Oracle allows you to define procedures that are implicitly executed when an INSERT, UPDATE, or DELETE statement is issued against the associated table. These procedures are called database triggers.

A trigger can include SQL and PL/SQL statements to execute as a unit and can invoke stored procedures. However, procedures and triggers differ in the way that they are invoked. While a procedure is explicitly executed by a user, application, or trigger, one or more triggers are implicitly fired (executed) by Oracle when a triggering INSERT, UPDATE, or DELETE statement is issued, no matter which user is connected or which application is being used.The figure below shows a database application with some SQL statements that implicitly fire several triggers stored in the database.



Triggers

Triggers are stored in the database separately from their associated tables. Triggers can be defined only on tables, not on views. However, triggers on the base table(s) of a view are fired if an INSERT, UPDATE, or DELETE statement is issued against a view.

How Triggers Are Used

In many cases, triggers supplement the standard capabilities of Oracle to provide a highly customized database management system. For example, a trigger can permit DML operations against a table only if they are issued during regular business hours. The standard security features of Oracle, roles and privileges, govern which users can submit DML statements against the table. In addition, the trigger further restricts DML operations to occur only at

certain times during weekdays. This is just one way that you can use triggers to customize information management in an Oracle database. In addition, triggers are commonly used to:

- automatically generate derived column values
- prevent invalid transactions
- enforce complex security authorizations
- enforce referential integrity across nodes in a distributed database
- enforce complex business rules
- provide transparent event logging
- provide sophisticated auditing
- maintain synchronous table replicates

Database triggers are defined on a table, stored in the associated database, and executed as a result of an INSERT, UPDATE, or DELETE statement being issued against a table, no matter which user or application issues the statement.
Oracle Forms triggers are part of an Oracle Forms application and are fired only when a specific trigger point is executed within a specific Oracle Forms application. SQL statements within an Oracle Forms application, as with any database application, can implicitly cause the firing of any associated database trigger.

Parts of a Trigger

A trigger has three basic parts:

- **A Triggering Event or Statement** :A triggering event or statement is the SQL statement that causes a trigger to be fired. A triggering event can be an INSERT, UPDATE, or DELETE statement on a table.
- **A Trigger Restriction** : A trigger restriction specifies a Boolean (logical) expression that must be TRUE for the trigger to fire. The trigger action is not executed if the trigger restriction evaluates to FALSE or UNKNOWN. A trigger restriction is an option available for triggers that are fired for each row. Its function is to control the execution of a trigger conditionally. You specify a trigger restriction using a WHEN clause.
- **A Trigger Action** : A trigger action is the procedure (PL/SQL block) that contains the SQL statements and PL/SQL code to be executed when a triggering statement is issued and the trigger restriction evaluates to TRUE. Similar to stored procedures, a trigger action can contain SQL and PL/SQL statements, define PL/SQL language constructs (variables, constants, cursors, exceptions, and so on), and call stored procedures. Additionally, for row trigger, the statements in a trigger action have access to column values (new and old) of the current row being processed by the trigger. Two correlation names provide access to the old and new values for each column.

Types of Triggers

When you define a trigger, you can specify the number of times the trigger action is to be executed: once for every row affected by the triggering statement (such as might be fired by an UPDATE statement that updates many rows), or once for the triggering statement, no matter how many rows it affects.

Row Triggers A row trigger is fired each time the table is affected by the triggering statement. For example, if an UPDATE statement updates multiple rows of a table, a row trigger is fired once for each row affected by the UPDATE statement. If a triggering statement affects no rows, a row trigger is not executed at all.
Row triggers are useful if the code in the trigger action depends on data provided by the triggering statement or rows that are affected. For example, Figure 15 - 3 illustrates a row trigger that uses the values of each row affected by the triggering statement.

Statement Triggers A statement trigger is fired once on behalf of the triggering statement, regardless of the number of rows in the table that the triggering statement affects (even if no rows are affected). For example, if a DELETE statement deletes several rows from a table, a statement-level DELETE trigger is fired only once, regardless of how many rows are deleted from the table.
Statement triggers are useful if the code in the trigger action does not depend on the data provided by the triggering statement or the rows affected. For example, if a trigger makes a complex security check on the current time or user, or if a trigger generates a single audit record based on the type of triggering statement, a statement trigger is used.

BEFORE vs. AFTER Triggers

When defining a trigger, you can specify the trigger timing. That is, you can specify whether the trigger action is to be executed before or after the triggering statement. BEFORE and AFTER apply to both statement and row triggers.

BEFORE Triggers BEFORE triggers execute the trigger action before the triggering statement. BEFORE triggers are used when the trigger action should determine whether the triggering statement should be allowed to complete. By using a BEFORE trigger for this purpose, you can eliminate unnecessary processing of the triggering statement and its eventual rollback in cases where an exception is raised in the trigger action. BEFORE triggers are also used to derive specific column values before completing a triggering INSERT or UPDATE statement.

AFTER Triggers AFTER triggers execute the trigger action after the triggering statement is executed. AFTER triggers are used when you want the triggering statement to complete before executing the trigger action. If a BEFORE trigger is already present, an AFTER trigger can perform different actions on the same triggering statement.

You can have multiple triggers of the same type for the same statement for any given table. For example you may have two BEFORE STATEMENT triggers for UPDATE statements on the EMP table. Multiple triggers of the same type permit modular installation of applications that

have triggers on the same tables. Also, Oracle snapshot logs use AFTER ROW triggers, so you can design your own AFTER ROW trigger in addition to the Oracle-defined AFTER ROW trigger.

You can create as many triggers of the preceding different types as you need for each type of DML statement (INSERT, UPDATE, or DELETE). For example, suppose you have a table, SAL, and you want to know when the table is being accessed and the types of queries being issued. Figure 15 - 4 contains a sample package and trigger that tracks this information by hour and type of action (for example, UPDATE, DELETE, or INSERT) on table SAL. A global session variable, STAT.ROWCNT, is initialized to zero by a BEFORE statement trigger, then it is increased each time the row trigger is executed, and finally the statistical information is saved in the table STAT_TAB by the AFTER statement trigger.

Trigger Execution

A trigger can be in either of two distinct modes:
Enabled An enabled trigger executes its trigger action if a triggering statement is issued and the trigger restriction (if any) evaluates to TRUE. **Disabled**A disabled trigger does not execute its trigger action, even if a triggering statement is issued and the trigger restriction (if any) would evaluate to TRUE.

When a trigger is fired, the tables referenced in the trigger action might be currently undergoing changes by SQL statements contained in other users' transactions. In all cases, the SQL statements executed within triggers follow the common rules used for standalone SQL statements. In particular, if an uncommitted transaction has modified values that a trigger being fired either needs to read (query) or write (update), the SQL statements in the body of the trigger being fired use the following guidelines:

- Queries see the current read-consistent snapshot of referenced tables and any data changed within the same transaction.
- Updates wait for existing data locks before proceeding.

Oracle internally executes a trigger using the same steps used for procedure execution. The subtle and only difference is that a user automatically has the right to fire a trigger if he/she has the privilege to execute the triggering statement. Other than this, triggers are validated and executed the same way as stored procedures.

Oracle automatically manages the dependencies of a trigger on the schema objects referenced in its trigger action. The dependency issues for triggers are the same as dependency issues for stored procedures. In releases earlier than 7.3, triggers were kept in memory. In release 7.3, triggers are treated like stored procedures; they are inserted in the data dictionary. Like procedures, triggers are dependent on referenced objects. Oracle automatically manages dependencies among objects.

Oracle Packages

According to the PL/SQL User's Guide and Reference, "A package is a schema object that groups logically related PL/SQL types, items and subprograms. "
Oracle's built-in packages dramatically extend the power of the PL/SQL language. A package is far more than just a way of logically grouping objects together.

The main reason packages are not more widely adopted is that users are unaware of the benefits they offer.

Benefits of Oracle Packages

1. Objects don't get **invalidated** when you makes changes to the body. That saves a lot of recompilation and makes changing the implementation much more painless. You will still have to recompile if you change the specification, but that's not something you should be doing very often.
2. You can "**overload**" subprograms (procedures/functions). You can have several subprograms with the same name, but with a different number of parameters, or different types. That is another thing that makes implementation changes more painless because you can keep legacy code if you like. You can also see the extra flexibility that offers developers.
3. You can have **persistent variables** throughout a session without storing anything in a database table. Packages can have variables and constants that are initialised when the packages is first used within a session, and then they are available for the remainder of the session for all future references to anything within that package. That comes in very handy.
4. Speaking of **initialisation**, being able to call a procedure automatically the first time a package is used within a session can also come in very handy.
5. You can take advantage of "**encapsulation**." In essence, you can hide the implementation details from users but still give them all the information they need to use the package. Since they aren't aware of the details, that means you can change them with minimal impact or risk. Packages also support private subprograms and variables which are available only to other subprograms within the package, and remain completely hidden and inaccessible to anything outside the package.
6. You may notice some **performance** improvement when using packages. When you first use a package, the entire package may be loaded into memory, meaning fewer disk I/Os as you use the related items within.

A package is a group of procedures, functions, variables and SQL statements created as a single unit. It is used to store together related objects. A package has two parts, Package

Specification or spec or package header and Package Body.

Package Specification acts as an interface to the package. Declaration of types, variables, constants, exceptions, cursors and subprograms is done in Package specifications. Package specification does not contain any code.

Package body is used to provide implementation for the subprograms, queries for the cursors declared in the package specification or spec.

Advantages:

- It allows you to group together related items, types and subprograms as a PL/SQL module.
- When a procedure in a package is called entire package is loaded, though it happens to be expensive first time the response is faster for subsequent calls.
- Package allows us to create types, variable and subprograms that are private or public

Oracle Utilities

The Oracle Database Utilities—comprising Oracle Data Pump and Oracle SQL*Loader—are a set of tools to allow fast and easy data transfer, maintenance, and database administration of Oracle databases.

Oracle Data Pump

Oracle Data Pump is a feature of Oracle Database 11g that enables very fast bulk data and metadata movement between Oracle databases. Oracle Data Pump provides new high-speed, parallel Export and Import utilities (expdp and impdp) as well as a Web-based Oracle Enterprise Manager interface.

- Data Pump Export and Import utilities are typically much faster than the original Export and Import Utilities. A single thread of Data Pump Export is about twice as fast as original Export, while Data Pump Import is 15-45 times fast than original Import.
- Data Pump jobs can be restarted without loss of data, whether or not the stoppage was voluntary or involuntary.
- Data Pump jobs support fine-grained object selection. Virtually any type of object can be included or excluded in a Data Pump job.

- Data Pump supports the ability to load one instance directly from another (network import) and unload a remote instance (network export).

SQL*Loader

SQL*Loader is a high-speed data loading utility that loads data from external files into tables in an Oracle database. SQL*Loader accepts input data in a variety of formats, can perform filtering, and can load data into multiple Oracle database tables during the same load session.

SQL*Loader provides three methods for loading data: Conventional Path Load, Direct Path Load, and External Table Load.

PL/SQL Object Types

Object-oriented programming is especially suited for building reusable components and complex applications. In PL/SQL, object-oriented programming is based on object types. They let you model real-world objects, separate interfaces and implementation details, and store object-oriented data persistently in the database.

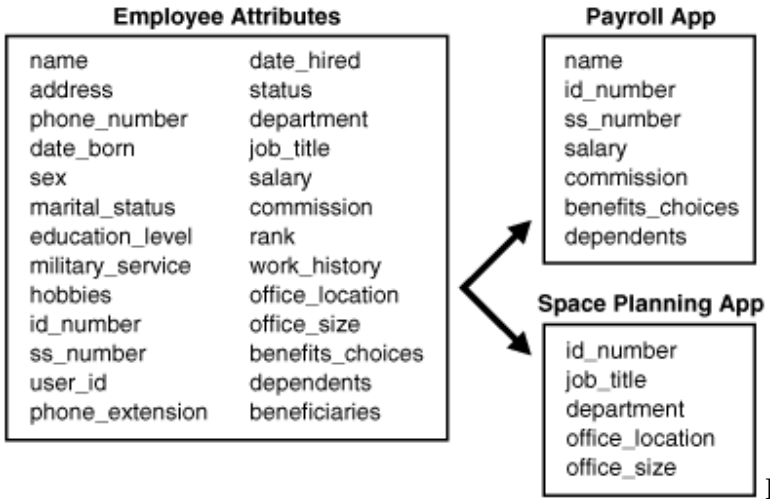
PL/SQL objects are named blocks of PL/SQL with declarative, executable and exception handling sections. The PL/SQL objects are stored in the data dictionary with a name and can be reused. PL/SQL objects include Packages, Stored Procedures, Functions and triggers.

An object type is a user-defined composite datatype representing a data structure and functions and procedures to manipulate the data. With scalar datatypes, each variable holds a single value. With collections, all the elements have the same type. Only object types let you associate code with the data.

The variables within the data structure are called attributes. The functions and procedures of the object type are called methods.

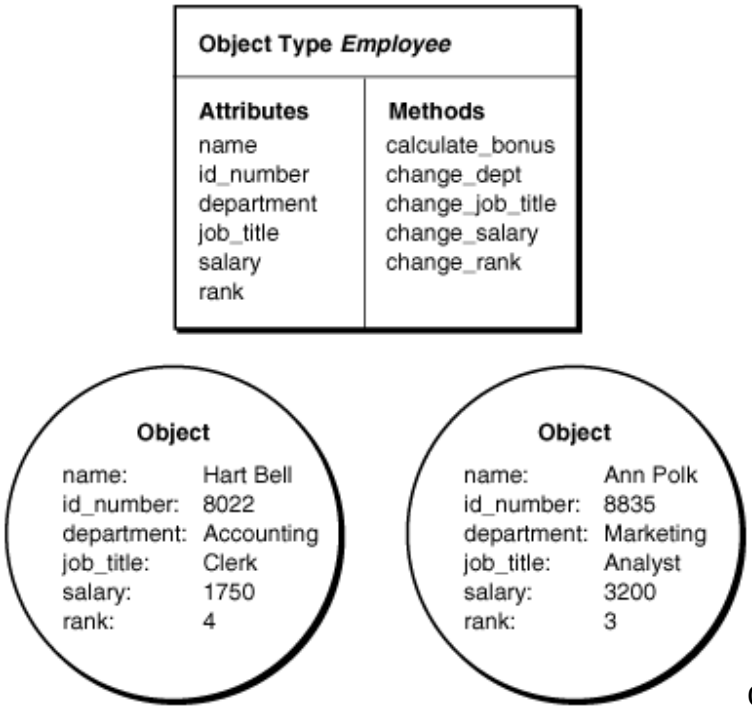
We usually think of an object as having attributes and actions. For example, a baby has the attributes gender, age, and weight, and the actions eat, drink, and sleep. Object types let you represent such real-world behavior in an application.

When you define an object type using the CREATE TYPE statement, you create an abstract template for some real-world object. The template specifies the attributes and behaviors the object needs in the application environment.



Each Application Uses a Subset of Object Attributes

Although the attributes are public (visible to client programs), well-behaved programs manipulate the data only through methodsthat you provide, not by assigning or reading values directly. Because the methods can do extra checking, the data is kept in a proper state.
At run time, you create instances of an abstract type, real objects with filled-in attributes.

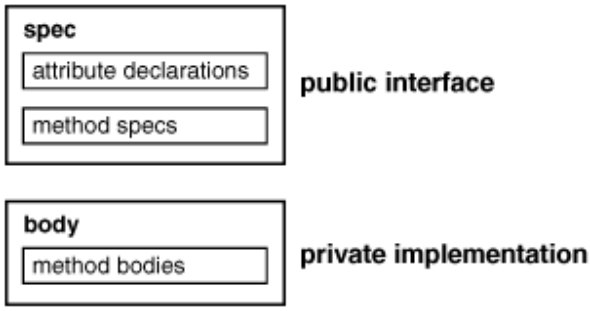


Object Type and Objects

(Instances) of that Type

Object types let you break down a large system into logical entities. This lets you create software components that are modular, maintainable, and reusable across projects and teams.
By associating code with data, object types move maintenance code out of SQL scripts and PL/SQL blocks into methods. Instead of writing a long procedure that does different things based on some parameter value, you can define different object types and make each operate slightly differently. By declaring an object of the correct type, you ensure that it can only perform the operations for that type. Object types allow for realistic data modeling.

Like a package, an object type has a specification and a body. The specification (or spec for short) defines the programming interface; it declares a set of attributes along with the operations (methods) to manipulate the data. The body defines the code for the methods.



Object Type Structure

All the information a program needs to use the methods is in the spec. You can change the body without changing the spec, and without affecting client programs. In an object type spec, all attributes must be declared before any methods. If an object type spec declares only attributes, the object type body is unnecessary. You cannot declare attributes in the body. All declarations in the object type spec are public (visible outside the object type).

Components of an Object Type

An object type encapsulates data and operations. You can declare attributes and methods in an object type spec, but not constants, exceptions, cursors, or types. You must declare at least one attribute (the maximum is 1000). Methods are optional.

Attributes
Like a variable, an attribute is declared with a name and datatype. The name must be unique within the object type (but can be reused in other object types). The datatype can be any Oracle type except:

- LONG and LONG RAW
- ROWID and UROWID

- The PL/SQL-specific types BINARY_INTEGER (and its subtypes), BOOLEAN, PLS_INTEGER, RECORD, REF CURSOR, %TYPE, and %ROWTYPE
- Types defined inside a PL/SQL package

You cannot initialize an attribute in its declaration using the assignment operator or DEFAULT clause. Also, you cannot impose the NOT NULL constraint on an attribute. However, objects can be stored in database tables on which you can impose constraints. The kind of data structure formed by a set of attributes depends on the real-world object being modeled. For example, to represent a rational number, which has a numerator and a denominator, you need only two INTEGER variables. On the other hand, to represent a college student, you need several VARCHAR2 variables to hold a name, address, phone number, status, and so on, plus a VARRAY variable to hold courses and grades.
The data structure can be very complex. For example, the datatype of an attribute can be another object type (called a nested object type). That lets you build a complex object type from simpler object types. Some object types such as queues, lists, and trees are dynamic, meaning that they can grow as they are used. Recursive object types, which contain direct or indirect references to themselves, allow for highly sophisticated data models.

Methods
In general, a method is a subprogram declared in an object type spec using the keyword MEMBER or STATIC. The method cannot have the same name as the object type or any of its attributes. MEMBER methods are invoked on instances, as in

instance_expression.method()

However, STATIC methods are invoked on the object type, not its instances, as in object_type_name.method()

Like packaged subprograms, methods have two parts: a specification and a body. The specification (spec for short) consists of a method name, an optional parameter list, and, for functions, a return type. The body is the code that executes to perform a specific task. For each method spec in an object type spec, there must either be a corresponding method body in the object type body, or the method must be declared NOT INSTANTIABLE to indicate that the body is only present in subtypes of this type. To match method specs and bodies, the PL/SQL compiler does a token-by-token comparison of their headers. The headers must match exactly.
Like an attribute, a formal parameter is declared with a name and datatype. However, the datatype of a parameter cannot be size-constrained. The datatype can be any Oracle type except those disallowed for attributes. The same restrictions apply to return types.

An object type can represent any real-world entity. For example, an object type can represent a student, bank account, computer screen, rational number, or data structure such as a queue, stack, or list. This section gives several complete examples, which teach you a lot about the design of object types and prepare you to start writing your own. Currently, you cannot define object types in a PL/SQL block, subprogram, or package.

You can define them interactively in SQL*Plus using the SQL statement CREATE TYPE.

PL/SQL supports a single-inheritance model. You can define subtypes of object types. These subtypes contain all the attributes and methods of the parent type (or supertype). The subtypes can also contain additional attributes and additional methods, and can override methods from the supertype.
You can define whether or not subtypes can be derived from a particular type. You can also define types and methods that cannot be instantiated directly, only by declaring subtypes that instantiate them.
Some of the type properties can be changed dynamically with the ALTER TYPE statement. When changes are made to the supertype, either through ALTER TYPE or by redefining the supertype, the subtypes automatically reflect those changes.
You can use the TREAT operator to return only those objects that are of a specified subtype.
The values from the REF and DEREF functions can represent either the declared type of the table or view, or one or more of its subtypes.