Basics

- ❑ You can run script using
- ❑ @script.sql or start script.sql at SQL> prompt
- ❑ @@script.sql pretends that I have changed the current directory to be that of currently executing file

Two types of variables:

```
Sql> DEFINE  sal =10000
Sql >DEFINE  sal     -- displays value of x
Sql> select * from employees where salary > &sal;
```

Bind variables are declared as

```
SQL> variable sal number
SQL> variable sal
variable   sal
datatype   NUMBER
SQL> begin
       :sal :=10000;
       dbms_output.put_line(:sal);
     end;

10000
```

Defines are always character strings expanded by SQL*Plus, and declared variables are used as true bind variables in SQL and PL/SQL.

```
variable count number
begin
   select count(*) into :count
   from employees;
end;
/
print :count
```

Language Fundamentals

- ❑ PL/SQL shares with ADA and PASCAL the additional definition of being a block-structured language.
- ❑ Inner block is also called as enclosed block, nested block, child block or sub block and outer PL/SQL block may be called as enclosing block or the parent block.
- ❑ A lexical unit in PL/SQL is any of the following:
  - Identifier
  - Literal
  - Delimiter
  - Comment
- ❑ Identifier must : Start with letter, can have up to 30 char, can include $, _ and #
- ❑ Many predefined functions and exceptions are declared in STANDARD and DBMS_STANDARD packages
- ❑ select * from v$reserved_words is listing all keywords
- ❑ Literal 10.5f is treated as float (binary_float 32 bit), 10.5d as double (binary_double 64 bit).

```
dbms_output.put_line( q'!abc'xyz!');    -- displays  abc'xyz
```

```
<<outer>>
declare
  v number := 1000;
begin
   declare
      v number := 20000;
   begin
      dbms_output.put_line(v);
      dbms_output.put_line(outer.v);
   end;
end;
```

Boolean can have three possible values:

```
declare
  v number := null;
  r boolean;
begin
    if v < 10000 then
```

```
        dbms_output.put_line('Less than 10000');
    else
        dbms_output.put_line('>= 10000');
    end if;
    r :=  v < 10000;
    case
     when  r is null then
        dbms_output.put_line('null');
     when  r  then
         dbms_output.put_line('true');
     else
         dbms_output.put_line('false');
         end case;
end;
```

```
Output :
>= 10000
Null
```

If a condition returns null then ELSE portion of IF is executed. To fix this problem, use NVL function.

PL/SQL uses short-circuit evaluation

When a condition's result is to be copied to a variable then PL/SQL doesn't short circuit condition if first condition is null. In this case, it proceeds to next condition to get the final value for condition.

```
declare
  v1 number := null;
  v2 number := 10;
  procedure print(r boolean) is
  begin
   case  r
     when true  then
         dbms_output.put_line('true');
     when false  then
         dbms_output.put_line('fale');
     else
        dbms_output.put_line('null');
   end case;
  end;
begin
    print( v1 < 10 and v2 > 20);  -- null and false
    print( v1 < 10 or  v1 > 20);   -- null or null
    print( v2 < 20 or v1 > 20);   -- true or null
    print( v2 < 20 and v1 is null);  -- true and true
end;
```

CASE raises an exception when no condition is satisfied and no ELSE is given.

```
declare
  v1 number := 10;
begin
  case
    when v1 > 100 then
       null;
    when v1 > 50 then
       null;
  end case;
end;
```

```
ORA-06592: CASE not found while executing CASE statement
```

For Loop rules

- ❑ Do not declare loop index
- ❑ Expressions used in range scheme are evaluated once when loops starts.
- ❑ You cannot change value of loop index. Errors : PLS-00363: expression 'I' cannot be used as an assignment target
- ❑ You can use loop label to exit outer loop from inner loop.

```
declare
begin
   <<iloop>>
   for  i in 1..10
   loop
       for j in 1..10
       loop
          dbms_output.put_line(' i = ' || i || ' j = ' || j);
          exit iloop when j > 5;  -- exit outerloop from inner loop using label
       end loop;
   end loop;
end;
```

EXCEPTION HANDLING

- Exceptions are declared in STANDARD package.
- For EXCEPTION_INIT error number cannot be -1403, instead you need to pass 100. It cannot be negative number less than -1000000

```
 declare
  invalid_month exception;
  pragma exception_init(invalid_month, -1843);
  v date;
begin
  v := to_date('11-abc-13');
exception
  when invalid_month then
     dbms_output.put_line('Not a valid month');
end;
```

Exception	**SQLCODE**
CURSOR_ALREDY_OPEN	6511
DUP_VAL_ON_INDEX	1
INVALID_CURSOR	1001
INVALID_NUMBER	1722
LOGIN_DENIED	1017
NO_DATA_FOUND	+100 (ORA-1403)
NOT_LOGGED_ON	1012
PROGRAM_ERROR	6501
STORAGE_ERROR	6500
TIMEOUT_ON_RESOURCE	51
TOO_MANY_ROWS	1422
TRANSACTION_BACKED_OUT	61
VALUE_ERROR	6502
ZERO_DIVIDE	1476

If exception is declared in package then its scope is every program whose owner has EXECUTE privilege on package.

When RAISE_APPLICATION_ERROR is invoked from a procedure then the following happens:

- Execution stops
- Changes made to OUT and INOUT arguments (without NOCOPY) will be reversed.
- Changes to package variables and database objects will NOT be rolled back.
- The error number range is -20000 to -20999.
- Error message can be up to 2K characters (extra chars ignored).
- Third parameter (KeepErrorStack) indicates whether you want to add the error to any already on the stack (TRUE) or replace it (default is false).

- WHEN OTHERS must be last among exception handlers
- SQLCODE returns error number. 0 if no error occurred.
- SQLERRM is a function that returns the error message for a particular error code. If you don’t pass error code, it uses SQLCODE to get error code.
- Maximum error message returned by SQLERRM is 512 bytes.

dbms_utility.format_error_backtrace
dbms_utility.format_error_stack
dbms_utility.format_call_stack

```
declare
   procedure p1
   is
      v date;
   begin
       v := to_date('11-abc-13');
   end;
begin
  dbms_output.put_line( sqlerrm(-1403));
  p1;
exception
  when others then
     dbms_output.put_line('BackTrace');
     dbms_output.put_line(dbms_utility.format_error_backtrace);
     dbms_output.put_line('Call Stack');
     dbms_output.put_line(dbms_utility.format_call_stack);
     dbms_output.put_line('Error Stack');
     dbms_output.put_line(dbms_utility.format_error_stack);
end;
```

```
ORA-01403: no data found
BackTrace
ORA-06512: at line 7
ORA-06512: at line 11

Call Stack
----- PL/SQL Call Stack -----
  object      line  object
  handle    number  name
23D4A4F4        18  anonymous block

Error Stack
ORA-01843: not a valid month
```

All user-defined exception have error code 1 and message “User Defined Exception”.

select sq.nextval from dual; Is using autonomous transaction. So even the enclosing transaction fails, it is not rolledback.

Any outstanding work performed by a statement that fails will be undone.

```
create or replace procedure changejob(p_empid number, p_jobid varchar) is
begin
   insert into logtable values(sysdate, 'Update ' || p_empid || ' with ' ||  p_jobid);
   update employees set job_id= p_jobid
   where employee_id = p_empid;
end;
```

```
set serveroutput on
declare
 d  date;
begin
   changejob(111,'IT_OG');
exception
   when others then
      select max(ed) into d
      from logtable;
      dbms_output.put_line(d);
      raise_application_error(-20111,'Invalid JOBID');
end;
```

The above program calls stored procedure, which inserts a row into LOGTABLE. As the block that calls stored procedure handles exception but raises application error, all changes made by stored procedure are undone.

If you don’t raise application error then the insert is not undone and transaction is still left uncommitted.

The following program displays ERROR as output.

```
set serveroutput on
declare
   e exception ;
begin
   declare
      e exception;
   begin
     if true then
         raise e;
     end if;
   end;
exception
 when others then
     dbms_output.put_line('Error');
end;
```

Data Types

- Oracle 11g LOB can hold up to 128 terabytes
- BINARY_FLOAT and BINARY_DOUBLE types support special value NaN (Not a number) as well as positive and negative infinity. Their application can lead to performance gains as arithmetic involving these binary types is performed in hardware whenever the underlying platform allows.
- SIMPLE_INTEGER, SIMPLE_FLOAT and SIMPLE_DOBLE are like BINARY_INTEGER, BINARY_FLOAT and BINARY_DOUBLE, but they do NOT allow NULL values and do NOT raise an exception when an overflow occurs.
- PLS_INTEGER and SIMPLE_INTEGER are PL/SQL specific. PLS_INTEGER is an integer with arithmetic implemented in hardware.
- For loop counters are PLS_INTEGERS
- PLS_INTEGER and BINARY_INTEGER are identical.
- SIMPLE_INTEGER is a subtype of PLS_INTEGER with NOT NULL constraint.

```
name datatype [NOT NULL] [:= | DEFAULT  default_assignment];
```

```
count  integer not null := 10;
count integer not null default 10;
```

```
constant_name CONSTANT datatype [NOT NULL] { := | DEFAULT } expression
```

- Anchored Declarations: %TYPE or %ROWTYPE.
- Anchor also establishes a dependency between the code and anchored element. If elements change then anchored elements are marked as INVALID.
- A NOT NULL constraint of one pl/sql variable is applied to another variable declared with %TYPE.
- A NOT NULL constraint from a database column is not automatically transferred to a variable that is declared with %TYPE.

```
declare
   v  employees.employee_id%type;   --  db not null is not applied to variable
   v1 number not null := 10;
   v2 v1%type;   -- error as it is also not not variable.
begin
  null;
end;
```

SUBTYPES

You can declare your own alias or subtypes to predefined data types.

```
SUBTYPE subtype IS basetype
```

Constrained subtype restricts subtype using RANGE option.

Subtype POSITIVE IS BINARY_INTEGER RANGE 1 .. 2147483647;

CAST function is used to convert one type to another.

```
Selet  Cast (hire_date as VARCHAR2(20)) from employees;

V : = Cast ( sysdate as varchar2);  -- do not specify size of target type.
```

- ❑ VARCHAR2 and CHAR can store up to 32767 bytes in PL/SQL. But they can store 4000 and 2000 respectively in SQL.
- ❑ You can declare VARCHAR2 either for bytes or characters.

```
Name  varchar2(10 char);
```

- ❑ Name occupies n no. of bytes for each char. In some character encoding it may be more than one byte for a char.
- ❑ Whether it takes byte or char depends on NLS_LENGTH_SEMANTICS init param, which is set to BYTE.
- ❑ STRING and VARCHAR are subtypes of VARCHAR2
- ❑ National char string is prefixed with n ' string ' and u ' string ' for Unicode code point.
- ❑ When you concatenate strings using CONCAT, if input strings are NON-CLOB then result is VARCHAR2, otherwise CLOB. If one is null then other one is returned, if both are null then null is returned.

```
CONCAT('abc',null)   →  abc
CONCAT(null,null)  → NULL
```

- ❑ Set NLS_COMP to LINGUISTIC to tell DB to use NLS_SORT for string comparison. NLS_SORT can be set to BINARY_CI to have case-insensitive search.

```
SUBSTR ('String', -2)   → Goes from end of the string towards beginning.   Output : ng
```

- ❑ Empty strings are considered to be NULL.
- ❑ CHAR variable is not considered NULL because they are never truly empty.

String Comparison

- ❑ If two are CHAR variables, then it uses blank-puddings comparison, where shorter of two strings is padded to longer value.
- ❑ If at least one string is VARCHAR then it uses non-blank-padding comparison.
- ❑ Literal are always considered as fixed-length CHAR datatype.
- ❑ NUMBER is implemented as a completely platform-independent fashion. It can stored from 1.0E-130 to 1.0E126-1.
- ❑ PLS_INTEGER (-2147383648 to +2177483647) and BINARY_INTEGER cannot be stored in database.
- ❑ Precision is 1 to 38, scale is -84 to 127.

DateTime Data Types

Date	Stores date and time resolved to second. Doesn't include time zone.
TimeStamp [(precision)]	Stores date and time resolved to billionth of a second (9 decimal places of precision).
TimeStamp [(precision)] With Time Zone	Timestamps + time zone
TimeStamp [(precision)] With Local Time Zone	Same as timestamp, but sensitive to time zone differences. Automatically converted to local time zone from database time zone. Stores value in Database time zone, but converts to local time zone.

- ❑ Precision refers to number of decimals allowed for faction of second. Default is 6 and max is 9.
- ❑ Timestamp(0) is like Date.
- ❑ TimeStamp takes up 7 bytes of storage (like DATE) when no subsecond is stored. With subsecond it takes up to 11 bytes.
- ❑ UTC - Coordinated Universal Time. Formally known as GMT (Greenwich Mean Time) or Zulu Time.
- ❑ PL/SQL will implicitly cast DATEs to TIMESTAMPs in subtraction expression.

Function	Time Zone	Return type
CURRENT_DATE	Session	Date
CURRENT_TIMESTAMP	Session	Timestamp with time zone
LOCALTIMESTAMP	Session	Timestamp
SYSDATE	Database	Date
SYSTIMESTAMP	Database	Timestamp with time zone
DBTIMEZONE	Database	Varchar2
SESSIONTIMEZONE	Session	Varchar2

Conversion function

```
TO_DATE(string [,format [, nls_language]])
TO_DATE(number [,format [, nls_language]])   -- Julian date to date
TO_TIMESTAMP(string [,format [, nls_language]])
TO_TIMESTAMP_TZ(string [,format [, nls_language]])
```

NLS_LANGUGAE is the language to be used to interpret names used.

- TZD is time zone daylight saving time
- TZR is time zone region
- TZH is time zone hours
- TZM is time zone minutes

```
select * from v$timezone_names   where tzabbrev = 'IST';
```

```
Alter session  set time_zone=timezone;  // Changes time zone of current session.
```

Interval data types

```
Interval year [(year_precision)] to month
Interval day [ (day_precision)] to second  [ (sec_precision)]
```

- Year_precision is 0 to 4 digits. Default is 2.
- Day_precision is 0 to 9. Default is 2.
- Frac_Sec_precision is 0 to 9. Default is 6.

You can convert number to interval with the following functions:

```
NUMTOYMINTERVAL   (called as num to Y M interval)
MUMTODSINTERVAL   (called as num to D S interval)
```

NOTE: YMINTERVAL_UNCONSTRAINED and DSINTERVAL_UNCONSTRAINED are interval data types with no loss of precision.

A string can be converted to Interval using:

```
TO_YMINTERVAL(‘Y-M’)
TO_DSINTERVAL(‘D HH:MI:SS.FF’)
```

You can add and subtract interval types provided they are of same type.

Interval literals

```
INTERVAL  ‘char’  start_element To end_element
```

```
ymi := INTERVAL ’40-4’ year to month;
ymi := INTERVAL ’40’  year;
dsi := INTERVAL ’10 1:1:2.10’  day to second;
```

```
declare
    d1 timestamp  := timestamp '2013-07-01 22:45:00.00';
    d2 date := to_date('1-jan-2013');
    d3 timestamp := to_timestamp('01-jan-2013 10:10:10.00 pm');
    iytm  interval year to month;
    idts  interval day(4) to second;

    d4 timestamp with time zone :=
          to_timestamp_tz('2013-07-01 22:45:00 -4:00','yyyy-mm-dd hh24:mi:ss tzh:tzm');
begin

    iytm := (d1 - d2) year to month;
    dbms_output.put_line(iytm);

    idts := (d1 - d2) day to second;
    dbms_output.put_line(idts);
    dbms_output.put_line(d4);

    dbms_output.put_line( extract( day from d1));
    dbms_output.put_line( extract( minute from idts ) );
end;
```

```
+00-06
+0181 22:45:00.000000
01-JUL-13 10.45.00.000000 PM -04:00
1
45
```

```
declare
   a interval year to month := numtoyminterval(20,'Month');
   b interval day to second := numtodsinterval(1000,'Minute');
   c interval day to second := to_dsinterval('10 10:20:30');
   d interval day to second := systimestamp  - sysdate;
begin
   dbms_output.put_line(a);
   dbms_output.put_line(b);
   dbms_output.put_line(c);
   dbms_output.put_line(d);
end;
```

```
+01-08
+00 16:40:00.000000
+10 10:20:30.000000
+00 00:00:00.888000
```

NLS_DATE_FORMAT specifies the default date format.
V$NLS_PARAMETERS contains information about NLS parameters.

Date and Timestamp literals

The following are allowed date and timestamp literals.

```
DATE ‘YYYY-MM-DD’
TIMESTAMP ‘YYYY-MM-DD HH:MI:SS[.FFFFFFFFF] [ {+|Z} HH:MI]’
```

CAST Function
Can be used to convert a string to any date time type.
PL/SQL does not allow size to be specified for target.

```
CAST ( source as target)
```

Extract Function
Used to extract a date component from datetime and interval.

```
EXTACT (component FROM  {datetime | interval})
```

RECORDS

- ❑ A record is a composite type that contains one or more members.
- ❑ Benefits : Data abstraction, aggregate operations, cleaner code
- ❑ Types : Table base records (JOBS%ROWTYPE), cursor based records (job_cur%ROWTYPE), and programmer-defined record.
- ❑ A record can be declared in declaration section or package specification.

```
TYPE  record_type  IS RECORD
(
   Member  type [ [NOT NULL]  :=|DEFAULT default_value],
   …
);

variable   record_type;
```

Type for member may be:

- Scalar type
- Programmer-defined subtype
- Anchored declaration using %TYPE and %ROWTYPE
- PL/SQL collection type
- REF CURSOR

Record level operations that are supported:

- Copy contents of one record to another as long as they are of same record type or compatible %ROWTYPE records (i.e. records have same number of fields and same or compatible datetypes)
- Set record to NULL
- Pass as parameter
- Return record from function

You can't do the following:

- Use IS NULL to see if all fields in the record have null values.
- Compare two records

```
declare
   type  job_rt is record
   (  job_id     jobs.job_id%type,
      job_title  varchar2(150),
      min_sal    number(5),
      max_sal    number(5)
   );

   job1 job_rt;
   job2 jobs%rowtype;

begin
   select * into job2   -- copy a row into record
   from jobs
   where job_id = 'IT_PROG';

   job1 := job2; -- fields need not have same name and sizes

   dbms_output.put_line( job1.job_title);
   dbms_output.put_line( job1.min_sal);

   job1.min_sal := 5000;

   -- update with record
   update jobs set row = job1
   where job_id = 'IT_PROG';

   job1.job_id := 'DBA';
   job1.job_title := 'Database Administrator';
   job1.min_sal := 10000;
   job1.max_sal := 20000;

   insert into jobs values job1;   -- INSERT with record
end;
```

COLLECTIONS

- Collection is like an array in tradition languages
- They allow you to maintain in-program lists of data
- Improve multirow SQL operations.
- Cache database information
- It is a single dimensional list of homogeneous elements.
- They are also called as arrays and tables
- You can create two-dimensional array by declaring collection of collections.
- Collection can be bounded (VARRAY) or unbounded (Associative Array and Table)
- Can be dense (if all elements between first and last element are defined and given value), or sparse (if rows are not defined and populated sequentially).
- Nested table and varray are can be stored in database. The table that host them is called as outer table.

Comparison of Collections

Varray	NestedTable	Associative Arrays(Index by tables)
1.Fixed Number of elements	Any number of elements	Any number of elements
2.Subscript will be sequence no	Subscript will be sequence no	Substring can be arbitrary number or string
3.Initialization required	Initialization required	Initialization Not required
4.Extend required	Extend required	Extend not required
5.Cannot be sparse	Can be sparse	NA
6.Can be stored in DB	Can be stored in DB	Cannot be stored in DB as a column

Associative Arrays

- They are single-dimensional, unbounded, spare collection of homogenous elements.
- Also known as index-by tables
- Index can be PLS_INTEGER or VARCHAR2
- In case of index by VARCHAR2, order of elements is decide by character set.
- INDEX BY clause can be any of the following;
    1. BINARY_INTEGER
    2. PLS_INTEGER
    3. POSITIVE
    4. NATURAL
    5. SIGNTYPE (-1,0, 1)
    6. VARCHAR2(32767)
    7. Table.column%TYPE
    8. Cursor.column%TYPE
    9. Package.variable%TYPE
    10. Package.subtype

Nested Tables

- Single-dimensional, unbounded collections of homogenous elements.
- Initially dense but can become sparse through deletions
- Can be defined both in PL/SQL and database
- They are multisets, which means there is no inherent order to the elements in the nested table.

```
declare
    type jobs_type is table of jobs%rowtype;
    jobs_tab jobs_type;
begin
    select *  bulk collect into jobs_tab
    from  jobs;

    for idx in  jobs_tab.first .. jobs_tab.last
    loop
          dbms_output.put_line( jobs_tab(idx).job_title);
    end loop;
end;
```

VARRAYS

- Single dimensional
- They are bounded and never sparse.
- Can be used in PL/SQL and the database
- The order is preserved
- BOOLEAN, NCHAR, NCLOB, NVARCHAR2, REF CURSOR, TABLE and VARRAY(non-sql datatype)
- ALTER TYPE type MODIFY LIMIT 100 INVALIDATE; modifies limit and invalidates all dependent object
- ALTER TYPE type MODIFY ELEMENT TYPE varchar2(100) CASCADE; Propagates the change to both the type and table dependents.

```
/* Associative array example */
declare
  type phones_aa is table of varchar2(10) index by varchar(20);
  phones phones_aa;
  idx    varchar2(20);
```

```
begin

    phones('srikanth') := 939333393;
    phones('praneeth') := 939334343;
    phones('padmaja') := 934343433;

    dbms_output.put_line( phones('padmaja'));

    -- print all

    idx := phones.first;
    while  idx is not null
    loop
       dbms_output.put_line( phones(idx));
       idx := phones.next(idx);
    end loop;

end;
```

MULTISET operators

Multiset operators combine the results of two nested tables into a single nested table.

- MULTISET EXCEPT
- MULTISET UNION
- MULTISET INTERSECT

```
--  Nested table demo
declare
    type phones_nt is table of varchar(20);
    phones phones_nt;
    officephones phones_nt;
    homephones phones_nt;
begin
    phones := phones_nt();  -- create nested table
    officephones := phones_nt('2222222222');

    phones.extend(3);

    phones(1) := '1111111111';
    phones(2) := '2222222222';
    phones(3) := '3333333333';

    homephones := phones multiset except officephones;

    for idx in  phones.first .. phones.last
    loop
       dbms_output.put_line( phones(idx));
    end loop;

    --  homephones
    for idx in  homephones.first .. homephones.last
    loop
       dbms_output.put_line( homephones(idx));
    end loop;

    phones.delete(2);  -- delete second items

    for idx in  phones.first .. phones.last
    loop
       begin
          dbms_output.put_line( phones(idx));
       exception
          when no_data_found then
            dbms_output.put_line('Item at ' || idx || ' is not found');
       end;
    end loop;
end;
```

```
-- VARRAY Example
declare
    type phones_va is varray(10) of varchar(20);
    phones phones_va;
begin
    phones := phones_va('111111','222222');

    phones.extend(1);
    phones(3) := '333333';

    for idx in  phones.first .. phones.last
    loop
       dbms_output.put_line( phones(idx));
    end loop;

    phones.trim;  -- delete last item

    for idx in  phones.first .. phones.last
    loop
          dbms_output.put_line( phones(idx));
    end loop;
end;
```

Collection Methods

Method	**Meaning**
COUNT	No. of elements in collection
DELETE[(stat [,end])]	Removes an element. With VARRAY, you can delete only the entire contents of the collection.
EXISTS	Returns true if element exists
EXTEND	Increases number of elements in VARRAY or NESTED TABLE
FIRST	Return smallest subscript
LAST	Returns largest subscript
LIMIT	Returns maximum no. of elements in VARRAY
PRIORT, NEXT	Return subscript that is before or next of the given subscript
TRIM	Removes element from the end of the collection.

- If COUNT is used with uninitialized nested table or VARRAY, it raises COLLECTION_IS_NULL predefined exception.
- You cannot DELETE an element from VARRAY. Only TRIM can be used. DELETE when used with VARRAY deletes all elements.
- EXTEND cannot be used with associative arrays.
- EXTEND(n,i) appends n number of elements. If I is given then copies value from ith element to appended elements. Mainly used with collection of NOT NULL nature.
- If VARRAY is extended beyond the limit then SUBSCRIPT_BEYOND_LIMIT exception is raised.

Collection pseudo-functions

CAST	Maps a collection of one type to another type
MULTISET	Maps a database table to collection.
TABLE	Maps a collection to database table. It is inverse of MULTISET.

- **SET**(collection) returns a unique collection
- Collection **IS A SET** return true if collection is unique
- Collection **IS NOT A SET** returns true if collection has duplicates.

```
declare
    type phones_nt is table of varchar(20);
    phones phones_nt;
    phones2 phones_nt;
begin
    phones := phones_nt();  -- create nested table
    phones.extend(3);
    phones(1) := '1111111111';
    phones(2) := '1111111111';
    phones(3) := '3333333333';
    if  phones is a set  then
        dbms_output.put_line('Yes.Unique SET');
    else
        dbms_output.put_line('Collection is not unique');
    end if;

    phones2 := set(phones);
    dbms_output.put_line( phones2.count);
end;
```

```
Insert into jobs values job_rec;
Update jobs set row = job_rec;
```

RETURNING

With this we can reduce network round trips, as it provides updated data after the change without we needing to query data again.

```
declare
    type jobs_table is table of jobs%rowtype;
    jobs_tab jobs_table;
    new_job jobs%rowtype;
begin
    select * bulk collect into jobs_tab from jobs;

    for idx in  jobs_tab.first..jobs_tab.last
    loop
        jobs_tab(idx).min_salary := jobs_tab(idx).min_salary  + 500;

        update jobs set row = jobs_tab(idx)
        where job_id = jobs_tab(idx).job_id
          returning job_id, job_title, min_salary, max_salary
          into new_job;

        dbms_output.put_line( new_job.min_salary);
    end loop;
end;
```

Restrictions on record-based updates and inserts :

- You can use record variable only on the right side of SET, in VALUES and in INTO of RETURNING
- ROW keyword must be used in UPDATE and no other values are allowed
- You cannot INSERT or UPDATE with a record that contains nested record or a function that returns nested record.
- You cannot use records in DML statement that are executed dynamically.

Autonomous Transaction

Can be defined for any of the following:

- Top-level (but not nested) anonymous PL/SQL block
- Functions and procedures defined in package or standard alone
- Methods of Object type
- Triggers

When To Use

- Logging mechanism
- Perform commits and rollbacks in database trigger
- Reusable application components
- Avoid mutating table trigger error for queries
- Call user-defined function in SQL that modify tables
- Retry counter

Rules

- If an autonomous transaction attempt to access resource help by main transaction then a deadlock can occur.
- To exit a program that has executed at least one DML, you must perform an explicit commit or rollback.
- You can have multiple commit or rollback statements inside your autonomous block.
- TRANSACTIONS parameter in init file specifies the max no. of transactions allowed concurrently in a session. Default is 75.
- Once changes are committed those changes are visible immediately in the main transaction. However, you can prevent it by using SET TRANSACTION ISOLATION LEVEL SERIALIZABLE.

Data Retrieval

Typical Query Operations

The following operations take place while executing a query.

Phase	**What is done?**
Parse	Ensures command is valid and determines execution plan
Bind	Associate value from your program with placeholders inside your SQL statement.
Open	Result set for SQL statement is determined. Pointer is set to first row.
Execute	Statement is run within SQL Engine.
Fetch	Retrieves next row.
Close	Closes cursor and release memory.

%BULK_ROWCOUNT
Used with FORALL returns the number of rows processed by DML execution.
%BULK_EXCEPTION
Used with FORALL, returns exception information that may have been raised by each DML execution.

- Select list (selected columns in SELECT) can contain PL/SQL variables and complex expressions.
- All implicit cursor attributes return NULL if no implicit cursor has yet been executed in the session.

```
CURSOR emp_cur RETURN  Employees%ROWTYPE
  Is select * from employees where department_id = 10;
```

- RETURN clause can specify table%ROWTYPE, anothercursor%ROWTYPE, or programmer-defined record.
- Cursor can be hidden in a package. Especially put cursor select in package body so that you can change SELECT without invalidating dependent object of that package.
- When you try to FETCH beyond end of cursor, PL/SQL will not raise exception and it won't do anything to alter the values of variables in INTO clause.

```
create or replace package job_pkg is
  cursor jobs_cur return jobs%rowtype;
end;

create or replace package body job_pkg is
  cursor jobs_cur return jobs%rowtype
    is
     select * from jobs;
end;


set serveroutput on
begin
    for jobrec in job_pkg.jobs_cur
    loop
        dbms_output.put_line( jobrec.job_title);
    end loop;
end;
```

You can assign default value for cursor parameters.

```
CURSOR emp_cur ( dept_id number := 10)
Is
query
```

- FOR UPDATE OF automatically obtains row-level locks on all rows identified by the SELECT statement.
- Rows remains locked until you COMMIT or ROLLBACK.
- If you do not include one or more columns after OF keyword, the database will lock all identified rows across all tables listed in the FROM clause.
- NOWAIT tell the database not to wait if the table is locked by another user. In this case control is immediately returned to your program.
- You can also use WAIT to specify maximum number of seconds the database should wait to obtain the lock.
- You cannot perform a FETCH on FOR UPDATE cursor after you COMMIT or ROLLBACK.

Cursor Variable

- Cursor variable is a reference to a cursor
- It provides mechanism for passing results of queries between PL/SQL programs.
- You can use standard cursor attributes like %ISOPEN, %FOUND etc. with cursor variables
- Cursor variable can be used in assignment
- If you assign one cursor variable to another then they both become aliases for the same cursor object.

- Two cursor variables are compatible if any of the following is true:
  - Both variables are of strong type and with same rowtype
  - Both variables are of weak type, regardless of rowtype_name
  - One variable is strong type and other is weak.
- A cursor object remains accessible as long as at least one active cursor variable refers to that cursor object.
- Cursor variables cannot be declared in package
- Cannot be passed in remote procedure calls
- Cannot be tested for equality, inequality or nullity using comparison operators
- Cannot assign NULL to cursor variable
- Cannot store in database
- Cannot be stored as an element of a collection

```
Type cursor_type_name IS REF CURSOR   [ RETURN return_type ];
```

```
Type emp_curtype  IS REF CURSOR RETURN emp%ROWTYPE;  -- Strong type.
TYPE curtype  IS REF CURSOR;    -- Weak type
My_cursor   SYS_REFCURSOR;   -- predefined weak type
```

```
declare
   type  job_curtype  is ref cursor return jobs%rowtype;
   jobcur  job_curtype;
   jobrec  jobs%rowtype;

   type  weak_curtype is ref cursor;
   weakcur weak_curtype;  -- or  weakcur SYS_REFCURSOR;
   v_job_title  jobs.job_title%type;
   v_firstname  varchar2(50);

   emp_cur  sys_refcursor;

   procedure  getemployees( p_emp_cur out sys_refcursor) is
   begin
        open p_emp_cur for select first_name from employees order by 1;
   end;
begin
   /* strongly typed cursor */
   open jobcur for select * from jobs;
   fetch jobcur into jobrec;
   dbms_output.put_line(jobrec.job_title);
   close jobcur;

   /* weakely typed cursor */
   open weakcur for select job_title from jobs order by job_title;
   fetch weakcur into v_job_title;
   dbms_output.put_line(v_job_title);
   close weakcur;


   getemployees(emp_cur);
   fetch emp_cur into v_firstname;
   dbms_output.put_line(v_firstname);
   close emp_cur;

end;
```

Cursor Expression

- Returns a nested cursor from within a query.
- It is denoted by CURSOR operator
- Can be used in explicit cursor declaration, dynamic SQL and REF CURSOR declarations and variables
- Cannot be used in implicit query.
- Cursor expressions can appear only in the outermost select list of query
- Can be placed in SELECT that is not nested in any other query expression, except when it is defined as a sub-query of the cursor expression itself.
- Cursor expression cannot be used when declaring a view
- Cannot perform BIND and EXECUTE operations on cursor expressions when using CURSOR expression in dynamic SQL.

```
declare
   cursor dept_employees is
     select department_name,
          CURSOR( select first_name from employees
                  where  department_id = d.department_id) as employees
     from departments d;

     empcur  sys_refcursor;  -- will hold employees details
     v_deptname varchar2(50);
     v_firstname varchar2(50);
begin
   open dept_employees;
   loop
       fetch dept_employees into v_deptname, empcur;
       exit when dept_employees%notfound;
       dbms_output.put_line( v_deptname);
       -- no need to open EMPCUR as it is already open for the current row
       loop
           fetch empcur into v_firstname;
           exit when empcur%notfound;
           dbms_output.put_line( lpad(v_firstname,30,'*'));
       end loop;
   end loop;
end;
```

Dynamic SQL

The following are the possibilities with dynamic SQL

- You can use DDL statements
- Support ad-hoc queries of web applications
- Softcode business rules and formulas

```
EXECUTE IMMEDIATE sql_string
   [ INTO  variables or record]
   [ USING [IN | OUT | IN OUT] bind_argument ] …
```

- Mode is relevant only for PL/SQL. Default is IN and that is the only mode for SQL.
- If sql_string ends with semicolon, it will be treated as a PL/SQL block.
- NDS supports all SQL datatypes. You may not bind values in the USING clause whose datatypes are specific to PL/SQL such as Boolean, associative arrays and user-defined records.
- You can use BULK COLLECT to retrieve multiple rows in dynamic query.
- We can use USING clause in OPEN FOR statement
- You cannot use bind variables for schema elements (tables, columns etc.) or entire chunks of SQL statement (such as WHERE clause).
- Binding is done after parsing and before execution.
- When you use RETURNING clause then OUT bind variables can be used.
- IN OUT and OUT bind variables are also used when calling PL/SQL programs dynamically
- You can bind values only to variables in PL/SQL block that have SQL type. For ex, Boolean cannot be used.
- For dynamic SQL, you must pass value for each placeholder even if they are duplicated.
- For dynamic PL/SQL, you must pass value for each unique placeholder.
- Passing null value is done either by un-initialized variable or TO_NUMBER(NULL) etc. functions.

```
EXECUTE IMMEDIATE  'update employees set salary = :sal where hire_date is null'
                                     using no_salary;
EXECUTE IMMEDIATE  'update employees set salary = :sal where hire_date is null'
                                     using to_number(null);
```

Methods

- No queries. DDL and DML with no bind variables.
- No Queries. DML with fixed number of bind variables
- Queries with fixed number of columns and bind variables, retrieving a single row or multiple rows.
- Statement with number of columns selected (for query) or the number of bind variables is not known until runtime.

Dynamic PL/SQL

- Must be a valid PL/SQL block

- Must start with DECLARE or BEING with END at the end
- It must end with ; otherwise is it not considered as PL/SQL
- Dynamic block can access only PL/SQL Code elements that have global scope (procedures, package).
- Dynamic PL/SQL blocks execute outside the scope of local enclosing block.
- Exception can be handled by local block in which the string was run with EXECUTE IMMEDIATE
- Dynamic PL/SQL is not treated as nested block.

DBMS_SQL Methods

VARCHAR2S	Maximum bytes per line is 256.
VARCHAR2A	Maximum bytes per line is 32676
DESC_TAB	Provides information about columns
DESCRIBE_COLUMNS	Retrieves column information
DEFINE_COLUMNS	Defines columns. Column numbers are used, not names
EXECUTE	Executes command
FETCH_ROWS	Fetches row
LAST_ROW_COUNT	Returns row count
COLUMN_VALUE	Gets value from a column
TO_REFCURSOR	converts a cursor number to weakly typed cursor variable
TO_CURSOR_NUMBER	converts a cursor variable (weak or strong) to cursor number

It is possible to avoid PARSE phase when you know that the SQL string is not changing and only bind variables are changing.

```
/* DBMS_SQL Example */
declare
  cmd  varchar2(100);
  cur  pls_integer;
  rows integer;
  title varchar2(30);
begin
   cur := DBMS_SQL.Open_cursor;
   cmd := 'select job_title from jobs';
   dbms_sql.parse(cur, cmd, DBMS_SQL.NATIVE);
   DBMS_SQL.DEFINE_COLUMN(cur,1,title,30);
   rows := DBMS_SQL.EXECUTE(cur);
   LOOP
       IF DBMS_SQL.FETCH_ROWS(cur)  > 0 THEN
         DBMS_SQL.COLUMN_VALUE(cur, 1, title);
         dbms_output.put_line(title);
       ELSE
         EXIT;
      END IF;
   END LOOP;
   dbms_sql.close_cursor(cur);
exception
   when others then
       dbms_sql.close_cursor(cur);
       dbms_output.put_line( sqlerrm);
end;
```

```
/* TO_CURSOR_NUMBER  example */
declare
  procedure execute_select(col_list varchar2)
  is
   cur  pls_integer;
   rows integer;
   curvar SYS_REFCURSOR;
   colcnt number;
   desctab dbms_sql.desc_tab;
   strvalue  varchar2(100);
  begin

   open curvar for  ' select '  || col_list || ' from jobs';
   cur := dbms_sql.to_cursor_number(curvar);
   dbms_sql.describe_columns(cur,colcnt,desctab);

   for i in 1..colcnt
   loop
```

```
         dbms_sql.define_column(cur,i,strvalue,200);
   END LOOP;

   while dbms_sql.fetch_rows(cur) > 0
    loop
      for i in 1..colcnt
      loop
        dbms_sql.column_value(cur,i,strvalue);
        dbms_output.put( strvalue || ' ');
      end loop;
      dbms_output.put_line('');
    end loop;

    dbms_sql.close_cursor(cur);

  exception
   when others then
        dbms_output.put_line( sqlerrm);
  end;

begin
   -- execute_select('job_id, job_title ');
   execute_select('job_title, max_salary ');

end;
```

```
/*  TO_REFCURSOR  Example */
declare
  phn dbms_sql.varchar2_table;
  phv dbms_sql.varchar2_table;

  procedure execute_select(p_cond varchar2, p_placeholders dbms_sql.varchar2_table, p_values
dbms_sql.varchar2_table)
  is
   cur  pls_integer;
   rows integer;
   l_names dbms_sql.varchar2_table;
   curvar SYS_REFCURSOR;
  begin
   cur := DBMS_SQL.Open_cursor;
   dbms_sql.parse(cur, 'select first_name from employees where ' || p_cond , DBMS_SQL.NATIVE);

   for i in  1..p_placeholders.count
   loop
        dbms_sql.bind_variable(cur, p_placeholders(i), p_values(i));
   end loop;

   rows := DBMS_SQL.EXECUTE(cur);

   curvar := dbms_sql.to_refcursor(cur);

   fetch curvar bulk collect into l_names;

   for i in 1..l_names.count
   loop
         dbms_output.put_line( l_names(i));
   END LOOP;
  exception
   when others then
        dbms_output.put_line( sqlerrm);
  end;
begin
   phn(1) := ':deptid';
   phn(2) := ':sal';

   phv(1) := '60';
   phv(2) := '5000';

   execute_select('department_id = :deptid and salary > :sal', phn, phv);

end;
```

Enhanced Security for DBMS_SQL

- Generation of unpredictable and randomized cursor numbers
- Restriction of use of DBMS_SQL whenever an invalid cursor number is passed to DBMS_SQL
- Rejection of DBMS_SQL operation when the current user attempting to use the cursor has changed from the user that opened the cursor.

```
declare
   type   job_table_type is table of  jobs%rowtype;
   j_tab  job_table_type;

   -- type job_ref_cur is ref cursor ;
   job_cur  SYS_REFCURSOR;
   job_rec  jobs%rowtype;
begin
/*
   execute immediate 'select * from jobs'
      bulk collect into j_tab;

   for i in  j_tab.first.. j_tab.last
   loop
       dbms_output.put_line( j_tab(i).job_title);
   end loop;

  */
   open job_cur for 'select * from jobs';

   loop
      fetch job_cur into job_rec;
      exit when job_cur%notfound;
      dbms_output.put_line(job_rec.job_title);
   end loop;


end;
```

Procedures, Functions and Packages

- AUTHID DEFINER | CURRENT_USER specify either definer rights or invoker rights
- Procedures can also use RETURN statement to terminate procedure.
- You cannot ignore return value of a function. It raises PLS-00221 error.
- If a function is terminated without returning a value then ORA-06503 error is raised.
- Parameters declaration must be unconstrained – VARCHAR2 and not VARCHAR2(20)
- PL/SQL allows a maximum of 64k parameters
- PLS-00230: OUT and IN OUT formal parameters may not have default expressions
- You cannot provide default value for OUT parameter
- When a PL/SQL procedure terminates with exception, values assigned to OUT parameters are not copied to actual parameters.
- You must list all of your positional parameters before any named notation parameters.
- Local modules must be located after all of the other declarations in declaration section.
- The order or precedence in establishing match for numeric parameter is : PLS_INTEGER or BINARY_INTEGER, NUMBER, BINARY_FLOAT, then BINARY_DOUBLE.

```
declare
   v  integer :=10;
   v2 binary_double :=20;
   procedure  p1(qty number)
   is
   begin
       dbms_output.put_line('p1  with number');
   end;
   procedure  p1(qty pls_integer)
   is
   begin
       dbms_output.put_line('p1 with pls_integer');
   end;
   procedure  p2(price pls_integer)
   is
   begin
       dbms_output.put_line('p2 with price pls_integer');
   end;

   procedure  p2(amount pls_integer)
   is
   begin
       dbms_output.put_line('p2 with amount pls_integer');
   end;
begin
   p1(10);
   p1(v2);
   p2 ( amount => v);
   -- p2 (100);

end;
```

Output :

```
p1 with pls_integer
p1  with number
p2 with amount pls_integer

If you include p2(100) then the following error :

PLS-00307: too many declarations of 'P2' match this call
```

PLSQL requires that you declare elements before using them in your code.

Requirements to call function from SQL

- ❑ Parameters must be IN mode
- ❑ Data types of parameters and return type must be recognized by Oracle Server.
- ❑ Function must be stored in database.
- ❑ Function may not modify database tables. However, this restriction is relaxed, if your function is defined with AUTONOMOUS transaction.
- ❑ When called remotely or through parallelized function, the function may not read or write the values of package variables.
- ❑ The function can update the values of package variables only if that function is called from SELECT list, VALUES or SET clause. If function is called in WHERE or GROUP BY, it may not write package variables.
- ❑ Named parameters can be used only from 11g.

```
create table logtable ( ed  date, entry varchar(200));

create or replace function GetTotal(p_deptno number)
return number  as
   PRAGMA AUTONOMOUS_TRANSACTION;
   v_total  number;
begin
   select sum(salary) into v_total
   from employees
   where department_id = p_deptno;

   if v_total is null then
       insert into  logtable values (sysdate,'No employees found for ' || p_deptno);
       commit;
   end if;
   return v_total;
end;


select department_name, gettotal(department_id)
from departments
```

DETERMINSTIC

- ❑ A function is considered to be deterministic if it returns the same result value whenever it is called with the same values for its IN and IN OUT arguments.
- ❑ No side effects
- ❑ System can use a saved copy of the function’s return result, if applicable.
- ❑ Used for functions that are called from SQL
- ❑ Functions used in function-based index and query of materialized view must be declared as DETERMINISTIC
- ❑ Improves performance

```
Function  fun( parameters) RETURN VARCHAR2  DETERMINISTIC
```

PARALLEL_ENABLE

Enables the function to be executed in parallel when called from within a SELECT statement.

RESULT_CACHE

Specifies that the input values and result of this function should be stored in the new function result cache.

Example for NOCOPY and Unhandled Exception

```
declare
   v number(5) := 10;
   w number(5) := 20;
   procedure  change(p1 out number, p2 out nocopy number)
   is
      lv number(5);
   begin
       lv := p1;
       p1 := 0;
       p2 := 0;
       raise  no_data_found;
   exception
      when others then
        null;
   end;
begin
   dbms_output.put_line(v);
   dbms_output.put_line(w);
   change(v,w);
   dbms_output.put_line(v);
   dbms_output.put_line(w);
exception
 when others then
   dbms_output.put_line('After error :');
   dbms_output.put_line(v);
   dbms_output.put_line(w);
end;
```

If you handle exception CHANGE then output :

```
10
20
0
0
```

If you don’t handle exception then output :

```
10
20
After error :
10
20

Note: NOCOPY hint may not be taken seriously.
```

Packages

- ❑ Package is initialized only once per session
- ❑ Package data is stored in UGA (User Global Area).
- ❑ Cursor variable are not allowed in package specification as they cannot be persisted.
- ❑ You can declare data structures like collection type, a record type, or a REF CURSOR
- ❑ You can specify AUTHID DEFINER or AUTHID CURRENT_USER. It must be in specification NOT body
- ❑ Data in the specification can be accessed from outside of package and it persists.
- ❑ Data in the body cannot be accessed from outside but it persists.
- ❑ The execution section of package BEGIN .. END is known as initialization section. This is executed for each session.
- ❑ Initialization section can be used for :
    - o Complex initialization logic
    - o Cache static session information
- ❑ Exception section handles any exceptions raised in the initialization section.

```
Package Body p1
Is
```

```
…
Begin
     …
Exception
    When… then
        …
End;
```

- If you declare a cursor in spec and define query in body then RETURN clause must specified.
- If you marks a package as SERIALLY_REUSABLE in spec and body then the duration of package data is reduced from a whole session to a single call of a program in the package.
- Global memory for serialized packages is allocated in SGA, not in UGA.

```
PACKAGE book_info
IS
   PRAGMA SERIALLY_REUSABLE;
   PROCEDURE  p1;
End;
PACKAGE BODY  book_info
IS
   PRAGMA SERIALLY_REUSABLE;
  ….
End;
```

Advantages

- Encapsulate data manipulation
- Avoid the hard-coding of literals
- Improve usability of built-in features
- Group together logically related functionality
- Cache session-static data to improve application performance

Associate Arrays

- `EXTEND` and `TRIM` cannot be used with associative arrays.

```
set serveroutput on
declare
   type  country_aa  is table of number(5)
     index by varchar2(50);
    countries  country_aa;
   country  varchar2(50);
begin
   countries('india') := 10000;
   countries('us') := 20000;
   countries('australia') := 4000;
   countries('spain') := 5000;

   country := countries.first; -- first index
   while  country is not null
   loop
       dbms_output.put_line( country  || ' - ' ||  to_char(countries(country )) );
       country := countries.next(country);  -- get next index
   end loop;
end;
```

output :

```
australia - 4000
india - 10000
spain - 5000
us – 20000
```

Triggers

Triggers are used to do the following:

- ❑ Perform validation
- ❑ Automate maintenance
- ❑ Apply rules concerning acceptable database administration activity in a granular fashion

Events that fire triggers:

- ❑ DML
- ❑ DDL
- ❑ Database Events like start up etc.
- ❑ INSTREAD OF
- ❑ Suspended statements

Transaction Participation:

- ❑ If a trigger raises an exception, that part of the transaction is rolled back.
- ❑ Any DML in trigger becomes part of main transaction
- ❑ You cannot issue commit or rollback from DML trigger. However, it is permitted, if you are using Autonomous transaction for trigger.

```
CREATE OR REPLACE TRIGGER BEF_INS_EMP
BEFORE INSERT ON EMPLOYEES
FOR EACH ROW
DECLARE
 PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN

END;
```

When a trigger is created with ERRORS then related DML operations fail.

```

create or replace trigger  error_trigger
before update on jobs
for each row
begin
   v:= 10;
end;
```

```
update jobs set min_salary = min_salary where job_id = 'IT_PROG';

SQL Error: ORA-04098: trigger 'HR.ERROR_TRIGGER' is invalid and failed re-validation
04098. 00000 -  "trigger '%s.%s' is invalid and failed re-validation"
```

WHEN clause

- ❑ Enclose the entire logical expression in side parentheses
- ❑ Do not include : in front of OLD and NEW names
- ❑ You can invoke built in functions only.
- ❑ You cannot invoke user-defined functions or function defined in built-in packages (DBMS_UTILITY etc.)
- ❑ WHEN can be used with only row-level triggers

NEW and OLD

- ❑ Both contain ROWID pseudo-column; The value is populated in both OLD and NEW with the same value in all circumstances.
- ❑ You cannot change values of OLD, but you can change values of NEW.
- ❑ NEW records can be changed only in BEFORE ROW triggers.
- ❑ You can use - REFERENCING OLD as old_emp NEW as new_emp - to rename OLD and NEW.

Points

- ❑ DELETING, INSERTING, UPDATING are used to determine the current operation
- ❑ In UPDATING ('sal') if column SAL is not present in the table then it returns false.
- ❑ A trigger can following another trigger for the same event using FOLLOWS option. This is shown as reference dependency in USER_DEPENDENCIES view.
- ❑ ROW level triggers cannot read or write the table from which it has been fired.
- ❑ Statement level triggers are free to read and write.

- ❑ Order for COMPOUND trigger is BEFORE STATEMENT, BEFORE EACH ROW, AFTER EACH ROW, AFTER STATEMENT.
- ❑ COMPOUND trigger can be used in FOLLOWS
- ❑ When a DML operation is completed all variables in COMPUND trigger are reset. This happens, even if there is any exception.

DDL Triggers

- ❑ You can define trigger that fire when DDL statements are executed.
- ❑ Example DDLs are CREATE TABLE, ALTER INDEX, DROP TRIGGER etc.
- ❑ USER_SOURCE data dictionary view does not get updated until after both BEFORE and AFTER DDL triggers are fired.
- ❑ If you define trigger on DDL (which matches all DDL statements) then you can use ORA_SYSEVENT to find out which event actually fired trigger.
- ❑ ORA_IS_ALTER_COLUMN function can be used to know which columns are being updated. Ex: ORA_IS_ALTER_COLUMN('salary')
- ❑ ORA_IS_DROP_COLUMN and ORA_IS_ALTER_COLUMN are unaware of which table is being used.

```
CREATE OR REPLACE TRIGGER trigger_name
{BEFORE | AFTER }
{DDL Event}
ON   {DATABASE | SCHEMA}
[ WHEN (…)]
DECLARE
..
BEGIN
     ..
END;
```

```
create or replace trigger alter_tab_trigger
after alter on schema
begin
   dbms_output.put_line('Altering the object :' ||  ORA_DICT_OBJ_NAME);
end;
```

```
alter table logtable  modify ( entry varchar2(250));
```

```
set serveroutput on
begin
   null;
end;
```

Note: In order to see the output of DBMS_OUTPUT used in DDL trigger, you must complete a PL/SQL block successfully.

Available attributes inside the body of the trigger:

Attribute Function	**What it returns**
ORA_DICT_OBJ_NAME	Name of the object affected by the firing DDL
ORA_DICT_OBJ_OWNER	Owner of the object
ORA_DICT_OBJ_TYPE	Type of the object affected by the firing DDL
ORA_LOGIN_USER	User

Some attributes return DBMS_STANDARD.ORA_NAME_LIST_T type, which is declared as follows:

```
TYPE ora_name_list_t  IS TABLE OF VARCHAR2(64);
```

Instead of triggers

WITH CHCEK OPTION is bypassed when updations are made from INSTEAD-OF trigger.

```
create or replace  view  cheap_employees
as
select * from employees
where salary < 5000
with check option;
```

// not allowed as it violates condition

```
update cheap_employees
  set salary = 6000
  where employee_id = 199;
```

```
create or replace trigger  is_cheap_employees_update
instead of update
on cheap_employees
for each row
begin
   update employees set salary = :new.salary
   where employee_id = :new.employee_id;
end;
```

// allows as updation is done through INSTEAD-OF trigger

```
update cheap_employees
  set salary = 6000
  where employee_id = 115;
```

Database Event Triggers

- STARTUP
- SHUTDOWN
- SERVERERROR
- LOGON
- LOGOFF
- DB_ROLE_CHANGE

```
CREATE [OR REPLACE] TRIGGER trigger_name
{BEFORE | AFTER} {event} ON {DATABASE | SCHEMA}
DECLARE
BEGIN
END;
```

The following combination is not allowed:

- BEFORE STARTUP
- AFTER SHUTDOWN
- BEFORE LOGON
- AFTER LOGOFF
- BEFORE SERVERERROR

In order to create startup event triggers, users must have been granted the **ADMINSTER DATABASE TRIGGER**.

AFTER SERVERERROR trigger will not fire when an exception is raised inside this trigger.

The following functions provide information about exception raised.

Function	**What it returns**
ORA_SERVER_ERROR	Error number
ORA_IS_SERVERERROR	Returns true if specified error number is in the current exception stack.
ORA_SERVER_ERROR_DEPTH	Returns number of errors on the stack.
ORA_SERVER_ERROR_MSG	Returns the full text of the error message at the specified position. It returns NULL if no error is found at the position.
ORA_SERVER_ERROR_NUM_PARAMS	Returns number of parameters associated with error message at the given position.
ORA_SERVER_ERROR_PARAM	Returns the value for the specified parameter position in the specified error.

SERVERERROR triggers are automatically isolated in their own autonomous transaction.

```
create or replace trigger error_logger
after servererror
on schema
declare
```

```
begin

   dbms_output.put_line('Errors :');
   dbms_output.put_line('========');
   for i in 1..ora_server_error_depth
   loop
       dbms_output.put_line(ORA_server_error_msg( i));
   end loop;
end;
```

```
set serveroutput on
begin
     dbms_output.put_line( to_date('31/31/2013'));
end;
```

- When INSTREAD OF trigger is used on nested table then you can use :PARENT record to access parent table's details.
- It is possible to create a trigger in DISABLED mode using DISABLE option. It is to be enabled later to be fired.
- Trigger views (USER_TRIGGERS) in the data dictionary do not display whether or not a trigger is in a valid state. This information is made available in USER_OBJECTS table.

Managing Code

Views that contains important information

USER_ARGUMENTS	Parameters for procedures and functions.
USER_DEPENDECIES	Dependencies to and from objects you own.
USER_ERRORS	Current set of compilation errors for all stored objects. This is used by SHOW ERRORS
USER_IDENTIFIERS	Contains all identifiers in your code. Populated by PL/Scope.
USER_OBJECTS	Objects owned by you
USER_OBJECT_SIZE	Size of objects you own.
USER_PLSQL_OBJECT_SETTINGS	Characteristics of a PL/SQL object that can be modified through ALTER and SET DDL commands, such as the optimization level, debug settings, and more.
USER_PROCEDURESS	Contains AUTHID and DETERMINISTIC details.
USER_SOURCE	Source code
USER_STORED_SETTINGS	PL/SQL Compiler flags.
USER_TRIGGERS USER_TRIG_COLUMNS	Triggers you own and columns identified with the triggers.

USER_OBJECTS

OBJECT_NAME	
OBJECT_TYPE	- PACKAGE, FUNCTION, TRIGGER, PROCEDURE
STATUS	- VALID, INVALID
LAST_DDL_TIME	- Last time object was changed.

USER_SOURCE

NAME	
TYPE	- Type of the object – PROCEDURE, FUNCTION, TRIGGER etc.
LINE	- Line number
TEXT	- Text

USER_OBJECT_SIZE

SOURCE_SIZE - Size of source in bytes.
PARSED_SIZE - Size of parsed form of the object in bytes. This must be in memory when any object that references this object is compiled.
CODE_SIZE -Code size in bytes. This must be in memory when object is executed.

USER_PLSQL_OBJECT_SETTINGS

PLSQL_OPTIMIZE_LEVEL	-Optimization level used to compile object
PLSQL_CODE_TYPE	- NATIVE or INTERPRETED
PLSQL_DEBUG	- Whether or not object was compiled for debugging
PLSQL_WARNINGS	- Compiler warning settings that were used to compile object

```
select name, plsql_optimize_level, plsql_code_type, plsql_debug, plsql_warnings
from user_plsql_object_settings;

ADD_JOB_HISTORY        2       INTERPRETED    FALSE   DISABLE:ALL
ALTER_TAB_TRIGGER      0       INTERPRETED    TRUE    DISABLE:ALL
AUTHOR_TYPE            2       INTERPRETED    FALSE   DISABLE:ALL
AUTHOR_TYPE            2       INTERPRETED    FALSE   DISABLE:ALL
CCTEST                 2       INTERPRETED    FALSE   ENABLE:INFORMATIONAL,DISABLE:PERFORMANCE,ENABLE:SEVERE
```

```
/* The following query displays all functions and their return types */
select object_name, data_type
from user_arguments
where position = 0


/* Triggers that have UPDATE (columns) clause but do not have WHEN clause*/
select trigger_name
from user_triggers ut
where when_clause is null
and exists ( select 1 from user_trigger_cols
             where trigger_name = ut.trigger_name)
```

```
alter session  set plscope_settings = 'IDENTIFIERS:ALL';

select  object_name , object_type from user_procedures

alter function gettotal compile;

select * from user_identifiers
where object_name = 'GETTOTAL' and type='VARIABLE' and usage = 'DECLARATION'
```

DEPENDENCY

- ❑ A dependency is a reference from a stored program to some database object outside the program.
- ❑ Server-based programs can have dependencies on tables, views, types , procedures, functions, sequences, synonyms, object types, package specifications.
- ❑ Program units are not dependent on package bodies and type bodies.
- ❑ Oracle does not use a compiled version of a program if any of the objects on which it depends have changed since it was compiled.
- ❑ Oracle 11g dependency tracking has increased from unit(table) to element (columns in table) within in the unit.
- ❑ Oracle tracks package specification and package body separately.
- ❑ Body depends on specification, but specification will never depend on its body.
- ❑ When your privileges on other schema objects are revoked then all objects that use that foreign object are also invalidated.
- ❑ When package body changes, dependents on package do not get invalidated.

```
/* list all objects that depend on EMPLOYEES table */
select name, type from user_dependencies
where referenced_name = 'EMPLOYEES'
```

```
execute deptree_fill('TABLE',user,'EMPLOYEES');
select * from ideptree;   /* shows recursively "referenced-by" query */
```

- ❑ Unless you use fully qualified variables in PLSQL inside your embedded SQL, when a new column to table is added with the same name as PL/SQL variable, the object is marked invalid.

```
create table t2( n1 number(10), n2 number(10));
create or replace procedure  t2_p1
is
   n3 number(5);
begin
   select count(*) into n3
   from t2;

   update t2  set n2 = n3;
end;

alter table t2  add ( n4 number(5));  /* this marks T2_P1 as INVALID */

select object_name, status from user_objects
where object_name like '%T2%';

OBJECT_NAME         STAUS
------------------- -------
T2_P1               INVALID
T2                  VALID
```

However, if you qualify variable then adding a column to table does not invalidate the object.

```
create or replace procedure  t2_p1
is
   n3 number(5);
begin
   select count(*) into n3
   from t2;

   update t2  set n2 = t2_p1.n3; -- qualifying object doesn't invalidate this object
end;
```

REMOTE DEPENDENCY

- When a procedure depends on remote object, local database doesn't attempt to invalidate the calling program in real-time when remote object changes.
- Local database defers checking until runtime
- PLSQL stored two kinds of information about each referenced remote procedure – its timestamp and signature.
- Timestamp is most recent date and time (down to second) when an object specification was reconstructed, as given by TIMESTAMP column in USER_OBJECTS view.
- For PLSQL program, TIMESTAMP is not necessarily the same as most recent compilation time because it is possible to recompile an object without reconstructing its specification.
- Signature is object name, datatype family, and mode of each parameter.
- Database uses either timestamp or signature, depending on the current value of REMOTE_DEPENDENCIES_MODE, which is set to TIMESTAMP by default
- Oracle recommends timestamp for server-to-server procedure call, and signature for client tools.
- Signature can cause false negatives – where runtime engine thinks that the signature hasn't changed, but it really has. Examples :
    - Changing only the default value of one of formal parameters.
    - Adding an overloaded program to an existing package. The caller will not bind to new version, even if it is supposed to.
    - Changing the name of the formal parameter and caller using named parameter notation.
- TIMESTAMP is prone to false positives, is immune to false negatives.
- When a remote procedure is modified and then local caller is called, oracle raises ORA-04062 error as local timestamp or signature (stored in local program) does not match remote procedure at runtime.
- If call to remote procedure is valid then on first call, local procedure is recompiled and if succeeds then its next invocation should run without error.
- There is no direct way for PLSQL program to use any of the following package constructs on remote server :
    - Variables
    - Cursors
    - Exceptions

Recompilation

- Automatic runtime recompilation - runtime engine will under many circumstances automatically recompile an invalid program before unit is called.
- ALTER.. COMPILE   Use ALTER command to recompile program unit.
- When a package is recompiled, other users using the package will get error as all session that started using package before its recompilation are not out of sync with package.

```
ALTER PACKAGE pkg COMPILE BODY RESUE SETTINGS;
```

REUSE SETTINGS ensure all compilation settings (optimization level, warning level, etc) previously associated with this program unit will remain the same.

Compile-Time Warnings

Three types of compile time warning:

Severe	Conditions that could cause unexpected behavior or actual wrong results, such as aliasing problems with parameters.
Performance	Passing VARCHAR2 to NUMBER column in UPDATE statement.
Informational	You might want to change to make code more maintainable.

```
ALTER SYSTEM SET PLSQL_WARNING = ‘ENABLE:ALL’;
```

Possible values after ENABLE are: ALL, SEVERE, PERFORMANCE, INFORMATIONAL

You can also treat a warning as an error using ERROR :<warning number>. Example:

```
ALTER SYSTEM SET PLSQL_WARNING=’ERROR:050005’;    --  converts  PLW-050005 to error.
```

DBMS_WARNING was designed to be used in scripts, where for a single program you want to treat warnings as errors etc.

If NOCOPY is not present for BODY and SPEC then Oracle applies whatever is found in SPEC. However it is a warning as there is a mismatch between body and spec – PLW-05000.

PLW-05001: Previous Use of String conflicts with this use:

```
alter session set plsql_warnings='ENABLE:ALL';
```

```
create or replace procedure p3
is
   v1 number(5);
   v1 date;
begin
   DBMS_OUTPUT.put_line('Two variables with same name ');
end;
```

PLW-05003: Same Actual Parameter at IN and NOCOPY may have side effects:

```
create or replace procedure sp2
is
     v1 varchar(105) :='First';
begin
     dbms_output.put_line(v1);
     sp1(v1,v1,v1);
     dbms_output.put_line(v1);
end;


create or replace procedure sp1(p1 in varchar, p2 in out varchar, p3 in out nocopy  varchar)
is
begin
     p2 := 'Second';
     p3 := 'Third';
end;

execute sp2
```

```
First
Second
```

PLW-05004: Identifier string is also declared in STANDARD or is a built-in

```
create or replace procedure sp1
is
  integer number(5);
begin
  integer := 10;
end;
```

- PLW:05005: Function string returns without value at line string
- PLW:06002: Unreachable Code
- PLW:07203: Parameter 'string' may benefit from use of NOCOPY compiler hint
- PLW:07204: Conversion away from column type may result in suboptimal query plan.
- PLW:06009: Procedure 'string' OTHERS handler does not end in RAISE or RAISE_APPLICATION_ERROR.

Wrapping PL/SQL Code

- A wrapped program is treated within the database just like a normal PL/SQL programs are treated.
- They cannot be seen in USER_SOURCE data dictionary view
- Wrapping makes reverse engineering of your code difficult.
- You cannot wrap the source code in triggers
- Wrapped code cannot be compiled into database of a version lower than that of wrap program.
- You cannot use SQL*PLUS substitution variables inside code that must be wrapped.
- You can wrap only package specification and bodies, object type specifications and bodies, and standalone functions and procedures.
- A program that is wrapped contains WRAPPED in its header.
- Wrapped code is much larger than the original source. The size of compiled code stays the same, although the time it takes to compile may increase.

WRAP executable

Program wrap.exe is placed in BIN directory.

```
WRAP iname=infile [oname=outfile]
```

- Default extension for infile is .sql
- If outfile is not given then it is taken as infile.plb, which stands for PL/SQL Binary.

DBMS_DDL.WRAP
Returns a string version of obfuscated version of your code.

DBMS_DDL.CREATE_WRAPPED
Compiles an obfuscated version of your code into the database.

```
declare
   lines dbms_sql.varchar2s;
begin
   lines(1) := 'create or replace procedure testobfus is begin dbms_output.put_line(100); end;';
   sys.dbms_ddl.create_wrapped(lines, lines.first, lines.last);
   dbms_output.put_line(
      sys.dbms_ddl.wrap('create or replace procedure testobfuscate is begin null; end;') );
end;
```

```
select * from user_source
where name  = 'TESTOBFUS';
```

OPTIMIZATION

Optimization settings are defined through PLSQL_OPTIMZE_LEVEL init parameter, which can be set to 0,1,2,3 (3 is only in 11g). The higher the number, the more aggressive is the optimization.

0	Turns off optimization.
1	Compiler will apply many optimizations to your code, such as eliminating unnecessary computations and exceptions. It will not, in general, change the order of your original source code.
2	Default value. More aggressive than level 1. Some changes might result in moving source code relatively far from original location. Compilation time may increase substantially.
3	Adds inlining of nested or local subprograms.

```
ALTER SESSION SET PLSQL_OPTIMIZE_LEVEL = 0;
ALTER PROCEDURE bigproc COMPILE PLSQL_OPTIMIZE_LEVEL=1;
ALTER PROCEDURE bigproce COMPILE REUSE SETTINGS;
```

All details are stored in USER_PLSQL_OBJECT_SETTINGS view.

Data Caching

- SGA caches the following information:
    - Parsed cursor
    - Data queried by cursors from database
    - Partially compiled representations of our programs
- SGA does not change program data.
- PGA is used to stored data related to program. It is specific to each connection.
- PLSQL programs can retrieve information more quickly from the PGA than it can from SGA.

Package-based caching

- Data related to package is cached for each session.
- The values stay in memory until your recompile your package, assign new values to variable, or disconnect

Deterministic Function Caching

- A function is considered to be deterministic if it returns the same result value whenever it is called with the same values for its IN and IN OUT arguments.
- The function has no side effects.
- Oracle can build a cache from the function's input and outputs.
- Declare a function as deterministic using DETERMINISTIC option after return type. Ex: RETURN VARCHAR2 DETERMINISTIC
- This cache is used only when function is called from SQL.
- This kind of cache is not used when function is called from PLSQL code.

RESULT CACHE

- Oracle stores both inputs and the return value in a separate cache for each function.
- It is not duplicated for each session.
- When function is called and cache is available then function is NOT called and cache is returned.
- Whenever changes are made to tables that are identified as dependencies for the cache, the database automatically invalidates the cache. Subsequent calls to the function will then repopulate the cache with consistent data.
- Starting from Oracle 11g R2, oracle will now automatically determines on which tables your returned data is dependent and correctly invalidate the cache when those tables contents change.
- If you rely on view then only view is to be listed and not base tables.
- Only function in schema and package can be associated with result cache.
- Cannot be used with PIPELINED function.
- Cannot be used if function has OUT or IN OUT parameters.
- Cannot be used any of the parameters are of type BLOB, CBLOB, NCLOB, REF CURSOR, collection, RRECORD, object type.
- Cannot be used if return type is any of the following : BLOB, CLOB, NCLOB, REF CURSOR, OBJECT TYPE, COLLECTION or Record that contains previously listed types.
- Cannot be used with function that have AUTHID CURRENT_USER.
- When checking to see if the function has been called previously with the same inputs, oracle considers NULL to be equal to NULL.
- User making change will bypass and see his changes. Other users see the data in the cache until the change is committed.
- When table on which it depends is marked invalid, and will need to be recompiled before it can be used.
- When function propagates an unhandled exception, the database will not cache the input values for that execution.

```
RESULT CACHE [RELIES_ON (table or view …)]
```

```
Create or replace function  f1 return varchar RESULT_CACHE …
Create or replace function  f1 return varchar RESULT_CACHE RELIES ON(employees)…
Create or replace function  f1 return varchar RESULT_CACHE RELIES ON(employees, jobs, depts)…
```

- A packaged function without RELIES_ON clause is needed in both spec and body.
- When RELIES_ON is used it may appear only in body and both use RESULT_CACHE.

- ❑ RESULT_CACHE_MAX_SIZE specifies the maximum amount of SGA memory that the function result cache can use.
- ❑ Oracle uses LRU algorithm to age out of cache
- ❑ DBMS_RESULT_CACHE package provides methods to manage contents of cache.
- ❑ V$RESULT_CACHE_STATISTICS, V$RESULT_CACHE_MEMORY, V$RESULT_CACHE_OBJECTS, and v$RESULT_CACHE_DEPENDENCY.

BULK Collect and FORALL

```
BULK COLLECT INTO collection_name [,collection_name]…
```

- ❑ You can use BULK collect from static SQL and dynamic SQL.
- ❑ You can use BULK collect in SELECT INTO, FETCH INTO and RETURNING INTO.
- ❑ Collection is populated starting from index 1 and inserts elements consecutively.
- ❑ You cannot use SELECT .. BULK COLLECT in FORALL
- ❑ BULK Collect doesn't raise NO_DATA_FOUND. Collection's count methods returns 0.
- ❑ LIMIT clause limits the number of rows fetched from bulk collect. But it can be used with FETCH and not with SELECT.

```
declare
    type jobs_table is table of jobs%rowtype ;

    type title_table is table of jobs.job_title%type;
    titles title_table;
    jobs_tab jobs_table;
begin
   select job_title bulk collect into titles
   from jobs order by job_title;

   for i in titles.first .. titles.last
   loop
       dbms_output.put_line( titles(i));
   end loop;


   select * bulk collect into jobs_tab
   from jobs order by job_title
   ;

   for i in jobs_tab.first .. jobs_tab.last
   loop
       dbms_output.put_line( jobs_tab(i).job_id );
  end loop;
end;
```

FORALL

Tells PL/SQL runtime engine to bulk bind into the SQL Statement all the elements of one or more collections before sending its statement to the SQL engine.

```
FORALL index IN
  [ lower_bound..upper_bound |
    Indices of indexing_collection |
    VALUES  of indexing_collection ]
[SAVE EXCEPTIONS]
Sql_statement;
```

Indexing_collection
Is PL/SQL collection used to select the indices in the bind array referenced in sql_statement;
INDICES OF and VALUES_OF are available options.

SAVE EXCEPTIONS
Is an optional clause that tells FORALL to process all rows; saving any exceptions that occur.

RULES:

- ❑ The body of FORALL must be single DML statement.
- ❑ The scope of index variable is FORALL statement and you may not reference it outside that statement.
- ❑ You cannot refer to this index outside the statement.

- ❑ Sparsely filled collections will raise error.
- ❑ Collection subscript referenced in the DML statement cannot be an expression. Ex: names(idx+2)
- ❑ %BULK_ROWCOUNT use the same subscripts or row numbers in the collections.
- ❑ DML Statements that raised the exception is rolled back to and implicit savepoint marked by PLSQL engine before execution of the statement. Changes to all rows already modified by that statement are rolled back.
- ❑ Any previous DML operations in that FORALL statement that already completed without error are NOT rolled back.
- ❑ If no specific action is taken (by adding SAVE EXCEPTIONS) the entire FORALL statement stops and remaining statements are not executed at all.

SQL%BULK_ROWCOUNT	Returns number of rows processed by each corresponding SQL Statement.
SQL%BULK_EXCEPTIONS	Returns a pseudo-collection that provides information about each exception raised in FORALL statement that includes SAVE EXCEPTIONS clause. ERROR_INDEX property returns row number and ERROR_CODE returns error.

```
declare
    type title_table is table of jobs.job_title%type
       index by pls_integer;
    type id_table is table of jobs.job_title%type
    index by pls_integer;

    titles title_table;
    ids id_table;
    newtitles title_table;
begin
   select job_id bulk collect into ids
   from jobs;

   forall indx in ids.first .. ids.last
      save exceptions
      update jobs
          set job_title = upper(job_title) || upper(job_title)
          where job_id = ids(indx)  and length(job_title) > 15
      returning job_title bulk collect into newtitles;

exception
when others then
   for i in 1.. sql%bulk_exceptions.count
   loop
      dbms_output.put_line( 'Error for ' ||  to_char(sql%bulk_exceptions(i).error_index)
       || ' is ' ||  sqlerrm( -1 *  sql%bulk_exceptions(i).error_code) );
   end loop;

   /*
   for i in newtitles.first .. newtitles.last
   loop
     if  sql%bulk_rowcount(i) <> 0  then
         dbms_output.put_line( 'Updated row at ' || to_char(i) ||  ' To ' ||  newtitles(i));
     end if;
  end loop;
  */
end;
```

INDICES OF Example:

```
declare
   type id_table is table of jobs.job_id%type   index by pls_integer;
   type indx_table is table of boolean    index by pls_integer;
   ids    id_table;
   indxs  indx_table;
begin
   indxs(1) := true;
   indxs(10) := true;
   ids(1) := 'IT_PROG';
   ids(10) := 'SA_MAN';
   forall i in INDICES OF indxs
      update jobs
          set job_title = upper(job_title)
          where job_id = ids(i);
end;
```

VALUES OF Example

```
declare
   type id_table is table of jobs.job_id%type
     index by pls_integer;
   type indx_table is table of pls_integer
      index by pls_integer;
   ids    id_table;
   indxs  indx_table;
begin
   indxs(1) := 1;
   indxs(2) := 10;
   ids(1) := 'IT_PROG';
   ids(10) := 'SA_MAN';
   forall i in VALUES OF indxs
      update jobs
          set job_title = upper(job_title)
          where job_id = ids(i);
end;
```

NOCOPY Parameter Mode Hint

- It requests that PL/SQL runtime to pass an IN OUT argument to be passed by reference rather than by value.
- By default IN is passed by reference and OUT and IN OUT are passed by value.
- Use NOCOPY after IN OUT or OUT
- The actual parameter for an OUT parameter under the NOCOPY hint is set to NULL whenever the subprogram containing the OUT parameter is called.
- NOCOPY is a hint and may be ignored by runtime.
- You can request NOCOPY only for entire structure(array) and NOT for an element.
- Some constraints like scale specification for numeric variable and NOT nULL constraints will result in NOCOPY being ignored.
- One or both records were declared using %ROWTYPE or %TYPE and the constraints on corresponding fields in these two records are different.
- NOCOPY is ignored in external or remote procedure call.
- If a program terminates with unhandled exception, you cannot trust the values in a NOCOPY actual parameter.

DBMS_OUTPUT

- Each user session has a DBMS_OUTPUT buffer of predefined size, which is set to UNLIMITED.
- This buffer is emptied when outermost PL/SQL block terminates.
- SET SERVEROUTPUT ON [SIZE UNLIMITED] enables output
- DBMS_OUTPUT.ENABLE (buffer_size => null);   sets buffer size to unlimited, otherwise buffer size is expressed in bytes.
- PUT method doesn't flush buffer. You need to use NEW_LINE to flush buffer.
- Largest string you can pass is 32767 bytes.
- Numbers and Dates are converted to VARCHAR, but Boolean cannot be converted.
- GET_LINE (string, status) is used to read a line. Copies line into string and status (0 for success) into status.
- GET_LINES(DBMS_OUTPUT.CHARARR, NoOfLines); If COUNT of array is 0 then it is end of it.

```
declare
  line varchar2(100);
  status  integer;
  line_count integer;
  lines dbms_output.chararr;
  procedure write is
  begin
   dbms_output.put_line('One');
   dbms_output.put_line('two');
   dbms_output.put_line('three');
  end;
begin

   write;
   /*
   -- read lines
   dbms_output.get_line(line,status);

   if status = 0 then
       dbms_output.put_line( ' Read : ' || line);
   else
       dbms_output.put_line('No line');
   end if;

   */

   line_count := 10;
   dbms_output.get_lines(lines, line_count);

   for i in 1 .. line_count
   loop
       dbms_output.put_line(lines(i));
   end loop;
end;
```

UTL_FILE

- This package is available only in SYS account.
- Grant privilege to other accounts to use it.
- R for Read, W for WRITE, A for APPEND.
- Append mode expects file to exists otherwise raises INVALID_OPERATION.
- FOPEN(FOPEN(location, filename, open_mode, max_line_size) return UTL_FILE.filetype
- IS_OPEN(file) RETURN BOOLEAN
- FCLOSE(file)   - raises WRITE_ERROR if buffered data is not written to file
- FCLOSE_ALL closes all files but doesn't mark file handles as closed so IS_OPEN with those handles still return true.
- GET_LINE(file, buffer OUT)
- PUT (file,string)
- NEW_LINE
- PUT_LINE(file, string, autoflush)   Default autoflush is false
- PUTF(file,format, args1, args2… args5); A maximum of 5 args.
- FFLUSH(file)
- FCOPY(src_location, src_file, dest_location, dest_filename, start_line, end_line)
- FREMOVE(location, file)
- FRENAME(src_location, src_filename,dest_location, dest_filename, overwrite);
- UTL_FILE.FGETATTR(location , filename , exists OUT BOOLEAN, file_length OUT NUMBER,   blocksize OUT NUMBER);

```
create or replace directory exam  as  'c:\1z0-144';

declare
   fh  sys.utl_file.file_type;
   line varchar2(100);
begin
   fh := sys.utl_file.fopen('EXAM', 'websites.txt','r');  --  EXAM name must be in upper
   loop
      sys.utl_file.get_line(fh,line,100);
      dbms_output.put_line(line);
   end loop;
Exception
  when no_data_found then
     sys.utl_file.fclose(fh);
end;
```

Write to file by taking data from JOBS table:

```
set serveroutput on
declare
   fh  sys.utl_file.file_type;
begin
   fh := sys.utl_file.fopen('EXAM', 'jobs.txt','w');  --  EXAM name must be in upper
   for jobrec in ( select * from hr.jobs)
   loop
      sys.utl_file.putf(fh,'Id : %s  Title : %s\n', rpad(jobrec.job_id,10), jobrec.job_title);
   end loop;
   sys.utl_file.fclose(fh);
end;
```

UTL_Mail

This package allows you to send mails.

1. Log into SYS account
2. Sql> start C:\oraclexe\app\oracle\product\11.2.0\server\rdbms\admin\utlmail.sql
3. Sql> start C:\oraclexe\app\oracle\product\11.2.0\server\rdbms\admin\prvtmail.plb
4. SQL> grant execute on utl_mail to public;
5. Log into HR
6. alter system set smtp_out_server= 'localhost'

```
begin
    Utl_mail.send( sender => 'webmaster@st.com',  recipients => 'james@st.com',
       subject => 'Test', message => 'Mail from Oracle');
end;
```

You can send mail to multiple users using :

```
    Recipients => ‘abc@gmail.com’, ‘xyz@yahoo.com’
```