The Complete PL/SQL Bootcamp

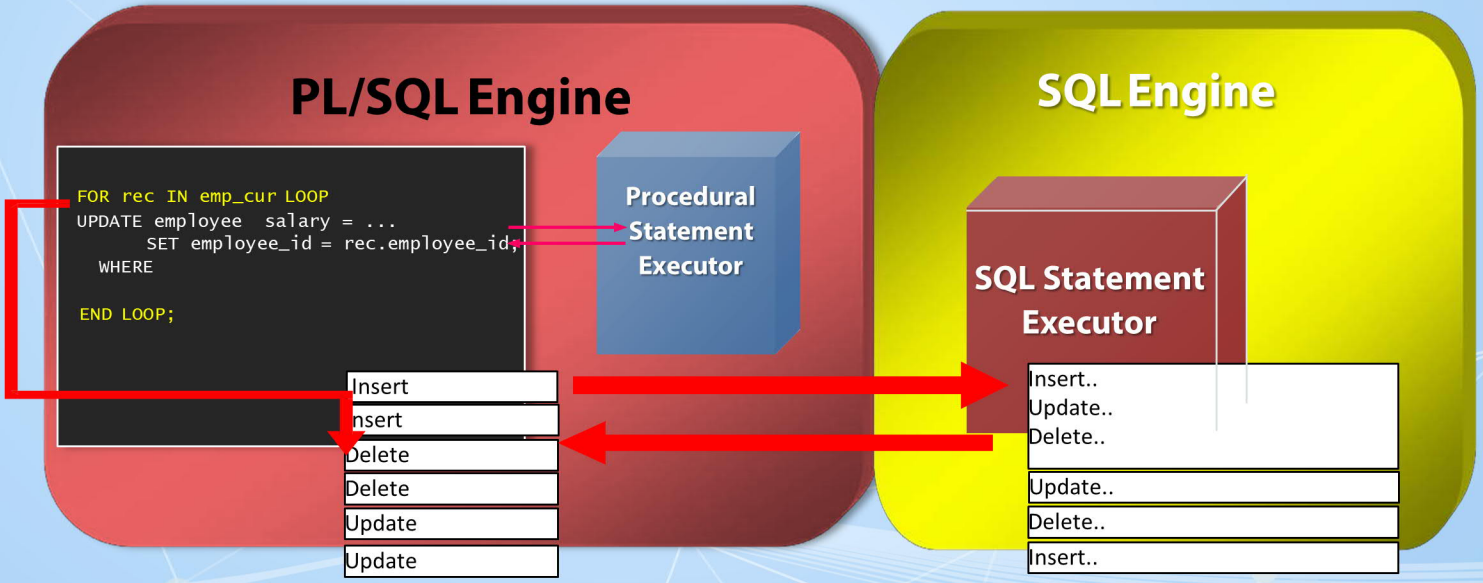
The Student Guide (2025)

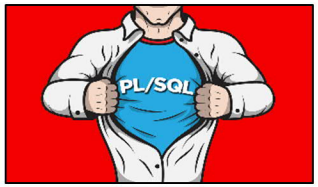
by Oracle Masters Training

What is PL/SQL ?

1. What is PL/SQL?

- PL/SQL stands for «**(P)**rocedural **(L)**anguage Extension to **SQL**»



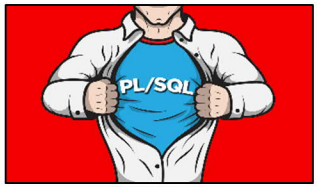


What is PL/SQL ?

- How will you have PL/SQL?

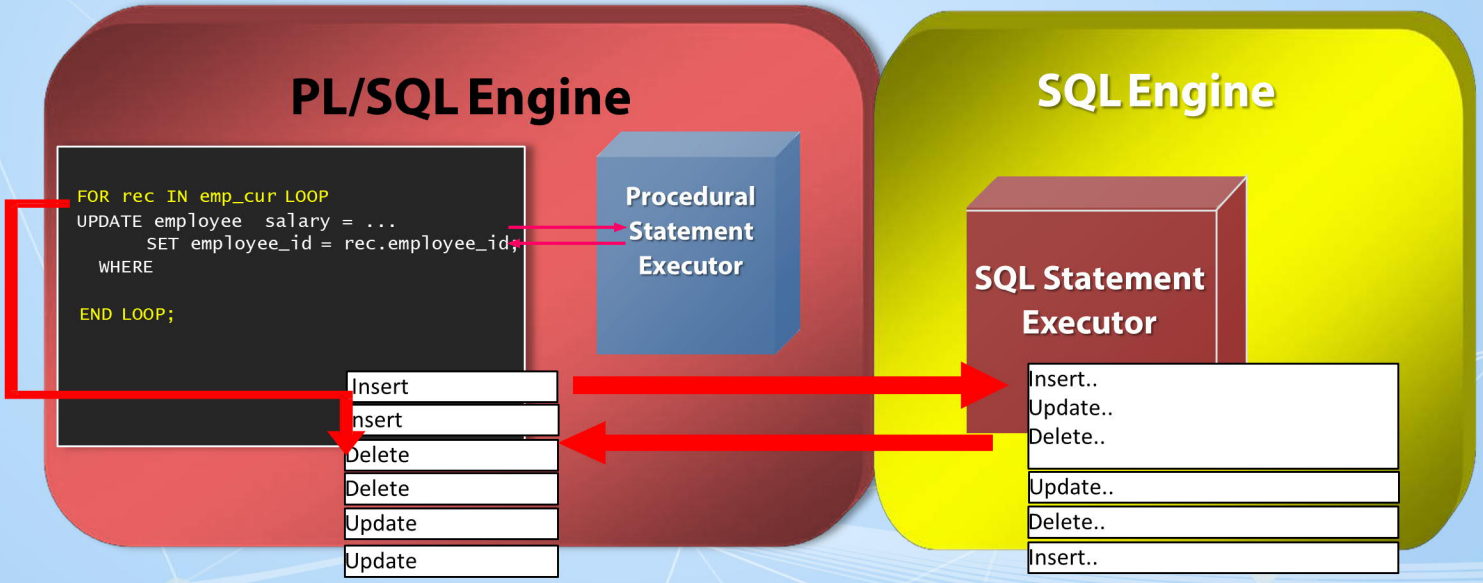
- Platform Independence

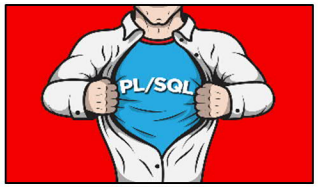
- So, Why PL/SQL?



PL/SQL Architecture

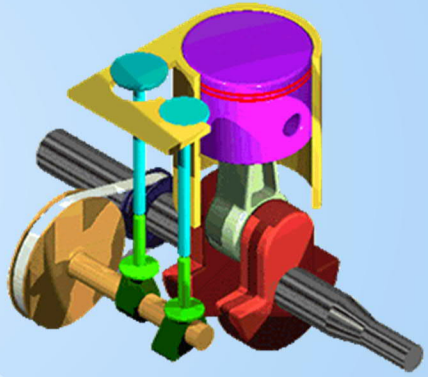
- Physical Architecture

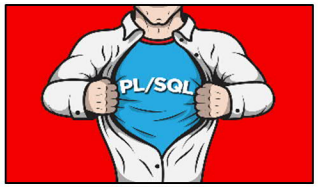




What is PL/SQL?

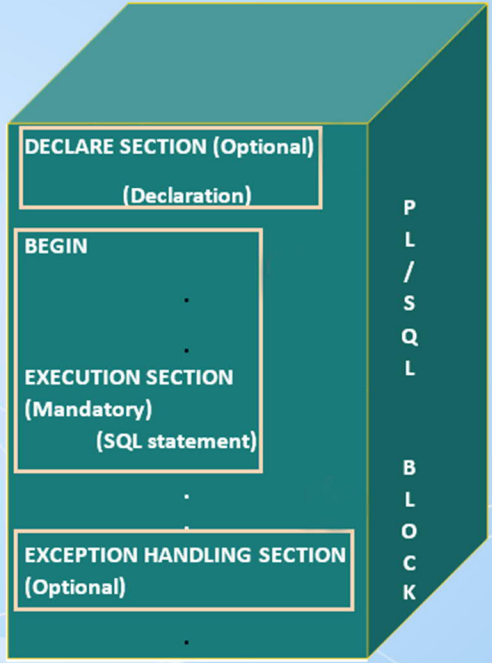
- **PL/SQL Logical Architecture**
  - Cooperates with SQL Engine
  - Enables Subprograms
  - Dynamic Queries
  - Case Insensitivity
  - Optimizer
  - Enables Object-Orinted Programming
  - Web Development
  - To sum up!..





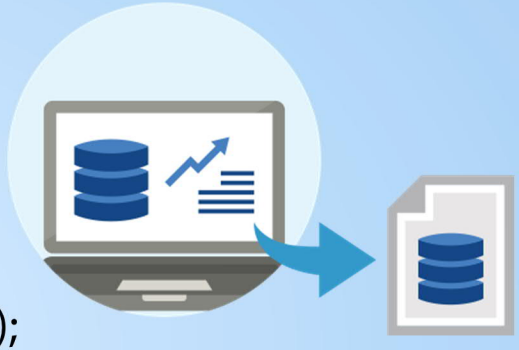
Let's Start Coding!

- **Blocks**
  - What are blocks?
    - DECLARE (Optional)
    - BEGIN (Mandatory)
    - EXCEPTION (Optional)
    - END; (Mandatory)
  - Three types of blocks
    - Anonymous Blocks
    - Procedures
    - Functions

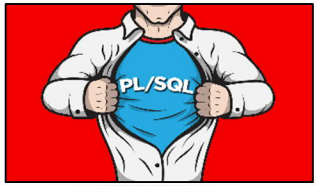


Let's Start Coding!

- **PL/SQL outputs**
  - Not an output language.
  - SET SERVEROUTPUT ON
  - DBMS_OUTPUT
  - DBMS_OUTPUT.PUT_LINE(*output_text*);



- Nested Blocks



Variables

- What are Variables & Why do We Need Them?
- PL/SQL Variable Types

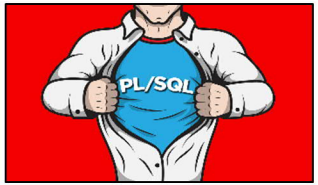
SCALAR

REFERENCE

LARGE OBJECTS

COMPOSITE

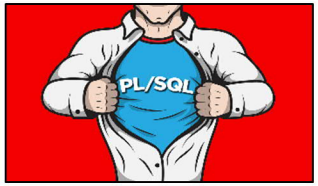
 |



Variables

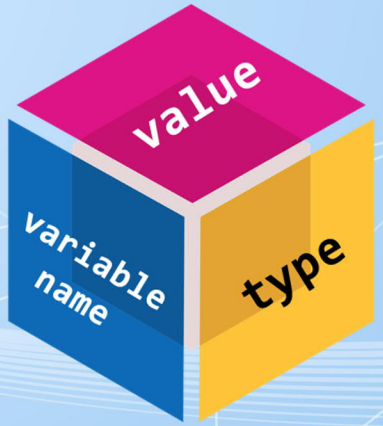
- Scalar Data Types

1. CHAR (max_length)
2. VARCHAR2(max_length)
3. NUMBER[precision,sale]
4. BINARY_INTEGER = PLSINTEGER
5. BINARY_FLOAT
6. BINARY_DOUBLE
7. BOOLEAN
8. DATE
9. TIMESTAMP (precision)
10. TIMESTAMP(p) WITH TIME ZONE
11. TIMESTAMP(p) WITH LOCAL TIME ZONE
12. INTERVAL(p) YEAR TO MONTH
13. INVERVAL(p) DAY TO SECOND(p)

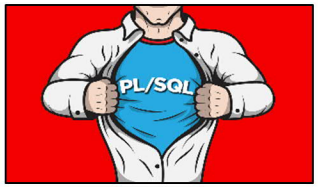


Variables

- **Variable Naming Rules**
  - Must start with a letter.
  - Can contain some special characters (like _ # $)
  - Can be maximum 30 characters.
  - Can not have Oracle's reserved words (select , varchar2 , etc)
- **Naming Conventions**
  - Why Naming Conventions?
    - VARIABLE – v_variable_name -- v_max_salary
    - CURSOR – cur_cursor_name -- cur_employees
    - EXCEPTION – e_exception_name – e_invalid_salary
    - PROCEDURE – p_procedure_name – p_calculate_salary
    - BIND VARIABLE – b_bind_name – b_emp_id



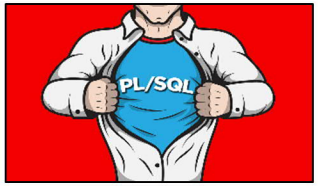
 |



Variables

- **Declaring & Initializing & Using Variables**
  - General Usage :

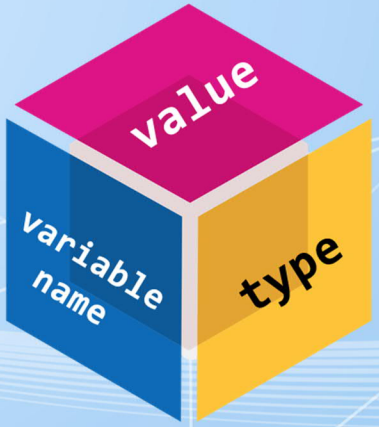
```
Name [CONSTANT] datatype [NOT NULL] [:= DEFAULT value|expression];
```

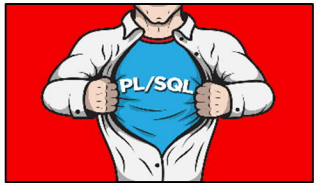


Variables

- **USING %TYPE ATTRIBUTE:**
  - What is %TYPE Attribute?
  - Why & Where to Use it?
  - How do We Use it?

```
name table.column_name%TYPE | another_variable%TYPE ;
```



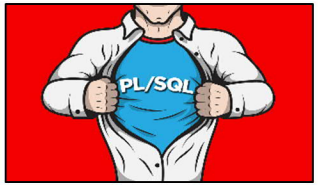


Variables

- **DELIMITERS & COMMENTING YOUR CODE.**
  - Delimiters are symbols with meaning.

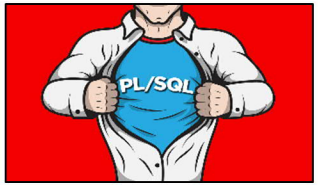
Symbol	Meaning
+	Addition
-	Subtraction | Negation
*	Multiplication
/	Division
=	Equality
@	Remote Access
;	Statement

Symbol	Meaning
<>	Inequality
!=	Inequality
||	Concatenation
:=	Assignment
--	Single Line Comment
/* */	Multi-Line Comment



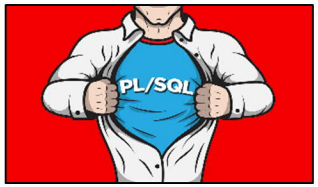
Variables

- **USING BIND VARIABLES**
  - Why Bind Variables?
  - Increases Performance
  - Created in the host environment
  - Its scope is the whole worksheet
  - Let's look at using bind variables by coding!



Variables

- **USING BIND VARIABLES.**
  - General Usage



Control Structures

IF STATEMENT

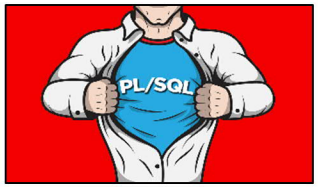
- What does Control Structures Mean?
- What is IF Statement?

```
IF condition THEN statements;
[ELSIF condition THEN statements;]
[ELSIF condition THEN statements;]
[ELSE  statements;]
END IF;
```

AND LOGIC TABLE			
AND	TRUE	FALSE	NULL
TRUE	true	false	null
FALSE	false	false	false
NULL	null	false	null

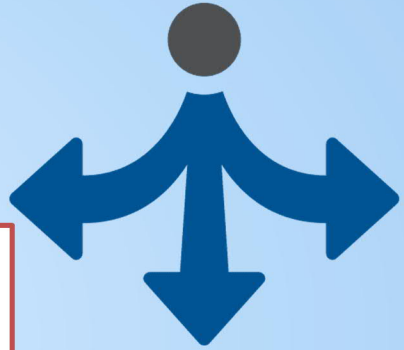
OR LOGIC TABLE			
OR	TRUE	FALSE	NULL
TRUE	true	true	true
FALSE	true	false	null
NULL	true	null	null

NOT LOGIC TABLE	
NOT	
TRUE	FALSE
FALSE	TRUE
NULL	NULL

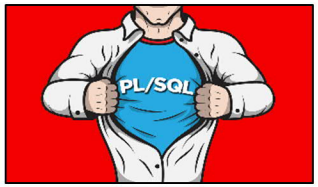


Control Structures

- **CASE EXPRESSIONS**
  - What is a Case Expression & Why to Use?
  - Case Expressions



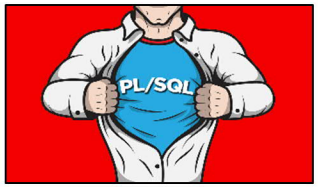
```
CASE [expression||condition]
        WHEN condition1 THEN  result1
        WHEN condition2 THEN  result2
                  ::::::::::
       [WHEN conditionN THEN  resultN]
       [ELSE                  result]
END;
```



Control Structures

- **LOOPS**
  - BASIC LOOPS

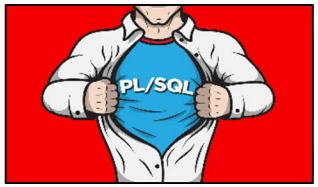
```
LOOP
        your_code;
                ::::::::::
        EXIT [WHEN condition];
END LOOP;
```



Control Structures

- **LOOPS**
  - WHILE LOOPS

```
WHILE condition LOOP
        your_code;
                ::::::::::
END LOOP;
```

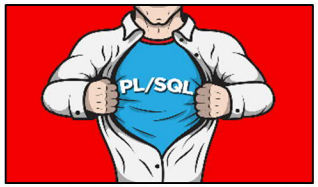


Control Structures

- **LOOPS**
  - FOR LOOPS

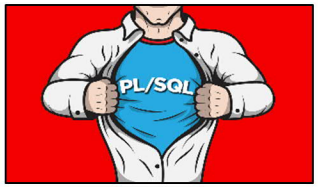
```
FOR counter IN [REVERSE]
        lower_bound..upper_bound LOOP
        your_code;
                ::::::::::
END LOOP;
```

- We cannot reach the counter outside of the loop.
- We cannot assign any value to the counter.
- Our bounds cannot be null.



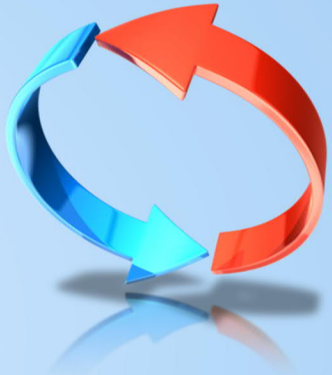
Control Structures

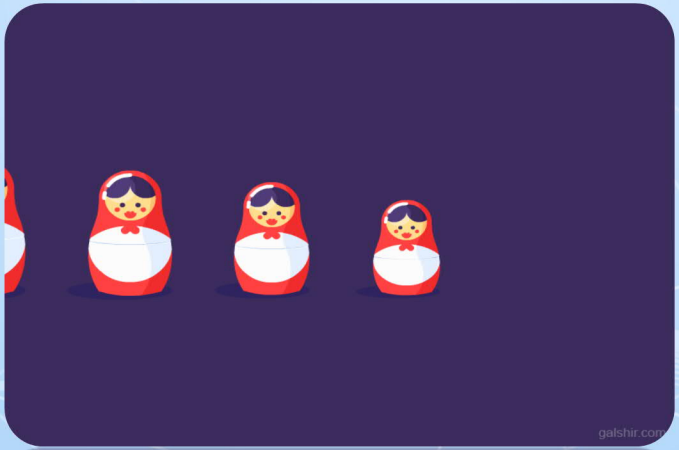
- **LOOPS – Recap**
  - **Which one to use?**
    - Use Basic Loops when..
    - Use While Loops when..
    - Use For Loops when..
  - To Sum Up !..

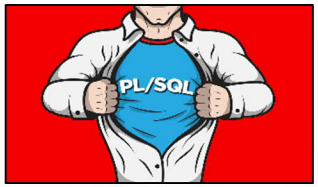


Control Structures

- **Nesting and Labeling the Loops**
  - We can nest loops..
  - We can use labels for loops.
  - Exting the inner loop will not exit the outer.
  - We can use labels to exit the outer loop.





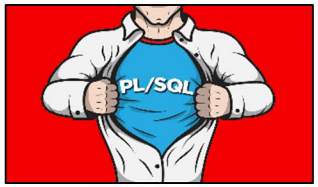


Control Structures

- **CONTINUE STATEMENT**
  - Why to use Continue Statement?
  - What are the benefits?
  - How is it used?

```
CONTINUE [label_name] [WHEN condition];
```

  - Let's make some examples..

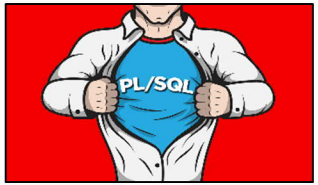


Control Structures

- **GOTO STATEMENT**
  - What is GOTO Statement?
  - Why to use GOTO Statement?
  - How is it used?

```
GOTO label_name;
```

  - There are some restrictions.
    - Into a control structure
    - Into an inner block from the outer
    - Out of a subprogram
    - In or Out of an Exception handler
  - Let's make some examples..

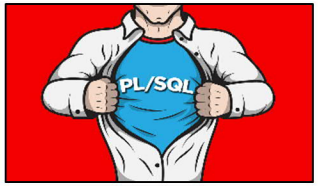


Using SQL in PL\SQL

- **OPERATING WITH SELECTED QUERIES**
  - Why to use SQL in PL/SQL?
  - There are some issues???
    - You cannot use DDL commands directly..
    - A block does not mean a transaction..
  - How to use SQL in PL/SQL?

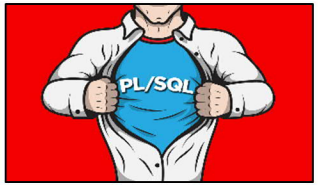
```
SELECT columns|expressions
INTO variables|records
FROM table|tables
[WHERE condition];
```

  - Naming Conventions & Ambiguities



Using SQL in PL\SQL

- **DML OPERATIONS IN PL/SQL**
  - We can use DML commands in PL/SQL code.
  - We can use PL/SQL Variables in a DML command.
    - *INSERT*
    - *UPDATE*
    - *DELETE*
    - *MERGE*

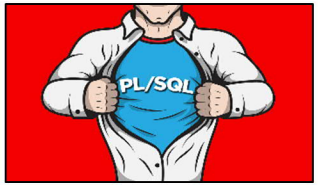


Using SQL in PL\SQL

- **USING SEQUENCES IN PL/SQL**
  - Why to use Sequences?
  - Where to use Sequences?
  - How to use Sequences?

```
SELECT sequence_name.nextval|currval INTO variable|column
FROM table_name|dual;
```

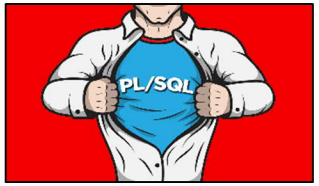
```
Variable|column := sequence_name.nextval|currval;
```



Composite Data Types

- **SIMPLE DATA TYPES vs COMPOSITE DATA TYPES**

```
declare
 v_name varchar2(50);
 v_salary employees.salary%type;
 v_employee_id employees.employee_id%type := 130;
begin
  select first_name ||' '|| last_name, salary into v_name, v_salary from employees
where employee_id = v_employee_id;
  dbms_output.put_line('The salary of '|| v_name || ' is : '|| v_salary );
end;
```

Composite Data Types

PGA

Records

rec_emp

id	first_name	last_name	salary	hire_date
101	Neena	Kocchar	17000	21-SEP-05

Nested Tables

c_emp_names

index	name
1	Luis
2	TJ
3	Nandita
4	William

Varrays

c_emp_names(4)

index	name
1	Luis
2	TJ
3	Nandita
4	William

Ass. Arrays

c_emp_names

index	name
113	Luis
132	TJ
184	Nandita
206	William

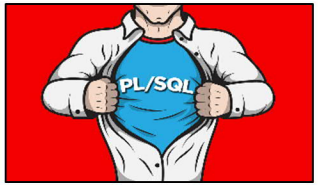
In Memory Tables

t_emps

index	first_name	last_name	salary	hire_date
101	Neena	Kocchar	17000	21-SEP-05
102	Lex	De Haan	17000	13-JAN-01
145	John	Russell	14000	01-OCT-04
104	Bruce	Ernst	6000	21-MAY-07

t_emps

index	last_name	salary	hire_date	department_id
Neena	Kocchar	17000	21-SEP-05	90
Lex	De Haan	17000	13-JAN-01	90
John	Russell	14000	01-OCT-04	80
Bruce	Ernst	6000	21-MAY-07	60



Composite Data Types

- **PL/SQL RECORDS**
  - What is a record?

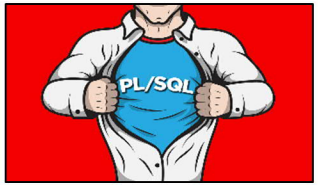
A Record

	Employee_id	first_name	last_name	salary	hire_date
rec_emp	101	Neena	Kocchar	17000	21-SEP-05

```
r_name TABLE_NAME%rowtype;
```

```
type type_name is record ( variable_name variable_type,
                           [variable_name variable_type,]
                           [..........]);
```

  - In Variable Types area, we can use:
    - ✓ All Valid PL/SQL Types, %TYPE, %ROWTYPE, NOT NULL and DEFAULT keywords.



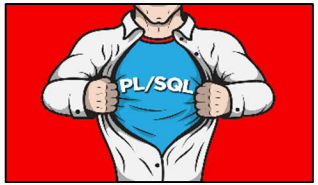
Composite Data Types

- **EASY DML WITH RECORDS**
  - How is the known way?

```
insert into departments values (280,'Temp Dept',null,1700);
```

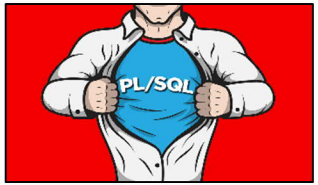
```
insert into departments (department_id, department_name) values (290,'Temp Dept2');
```

  - Why is it useful?
  - Let's make examples..



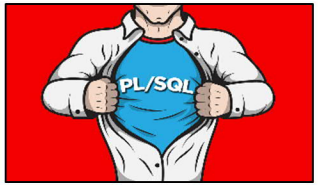
Composite Data Types

- **WHAT ARE COLLECTIONS?**
  - The second type of the composite data types.
  - Why do we need a collection?
  - Collections have a list of the same type.
  - There are 3 types of collections..
    - Nested Tables
    - VARRAYs
    - Associative Arrays
  - TABLE Operator



Composite Data Types

- **VARRAYS**
  - Identical of Arrays in other languages.
  - Upper limit is certain.
  - About Varrays :
    - Varrays are bounded
    - It's index start from 1
    - Varrys are one dimentional arrays
    - Varrays are null by default
  - Let's make examples..

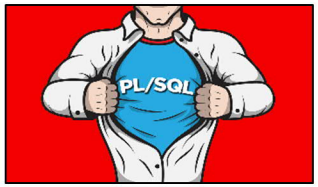


Composite Data Types

- **NESTED TABLES**
  - Very similar with the varrays.
  - Compare to varrays..
    - ✔ Key-Value Pairs
    - ✔ 2 GB at most
    - ✘ We can delete any values
    - ✘ Not stored consecutively
    - ✘ Nested tables are unbounded

```
type type_name as table of
     value_data_type [NOT NULL];
```

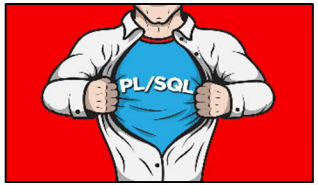
```
create or replace type type_name as table of
     value_data_type [NOT NULL];
```



Composite Data Types

- **ASSOCIATIVE ARRAYS**
  - Known as Index By Tables
  - Compare to other collections..
    - ✗ The key can have a string
    - ✗ Keys does not need to be sequential
    - ✓ Can have scalar & record types
    - ✗ Do not initialize associative arrays
    - ✗ Associative arrays are indexed

```
type type_name as table of value_data_type [NOT NULL]
INDEX BY {PLS_INTEGER | BINARY_INTEGER |VARCHAR2(size)};
```

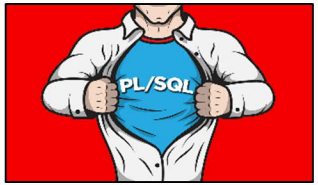



Composite Data Types

- **STORING COLLECTIONS IN TABLES**
  - We can store varrays and nested tables in database tables.
  - They are stored with different methods
  - Store and use easier-faster
  - Our types must be schema level

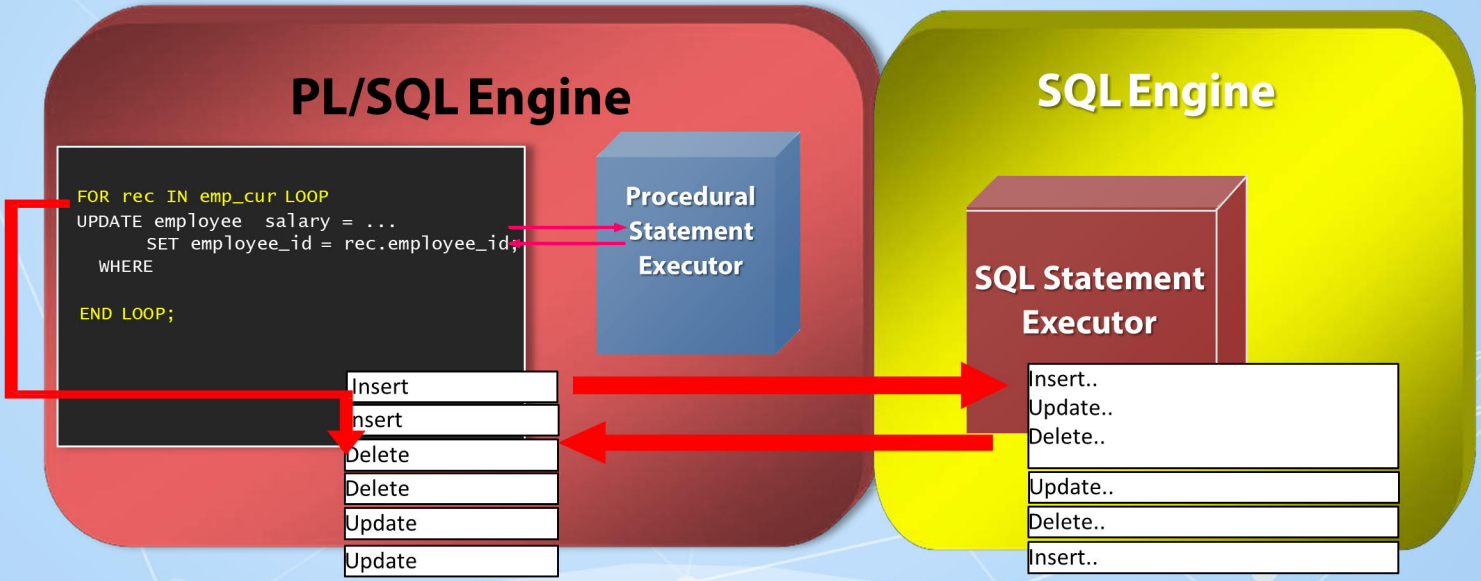
Employees table

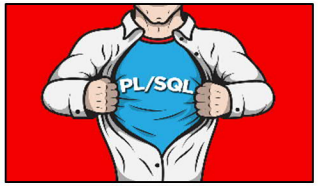
EMP_ID	FIRST_NAME	LAST_NAME	PHONE_NUMBER	
101	Bob	Green	Home	'111.111.1111'
			Work	+1-541-754-3010
102	David	Brown	Home	'111.111.1111'
			Work	+1-541-754-3010
			Mobile	+1-541-754-3510
			Fax	+1 (408) 999 8888



Cursors

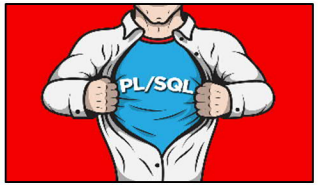
- **WHAT ARE CURSORS & CURSOR TYPES?**





Cursors

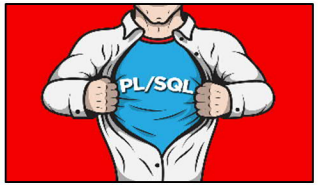
- **WHAT ARE CURSORS & CURSOR TYPES?**
  - Cursors are pointers to the data
  - There are two types of cursors.
    - Implicit Cursor
    - Explicit Cursor
  - We can control the cursors
  - Explicit cursors are created by the programmers
  - Collections vs Cursors
  - You cannot go back in cursors



Cursors

- **USING EXPLICIT CURSORS**
  - The Cursor Usage :
    - Declare
    - Open
    - Fetch
    - Check
    - Close

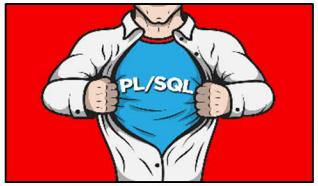
```
 declare
   cursor cursor_name is select_statement;
begin
   open cursor_name;
   fetch cursor_name into variables,records etc;
   close cursor_name;
end;
```

Cursors

- **CURSORS WITH PARAMETERS**
  - Cursors can be used for many times
  - We can use cursors with parameters

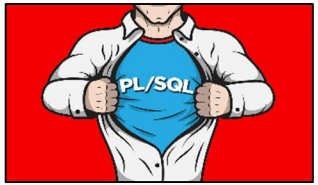
```
 declare
   cursor cursor_name(parameter_name datatype,…)
   is select_statement;
begin
   open cursor_name(parameter_values);
   fetch cursor_name into variables,records etc;
   close cursor_name;
end;
```



Cursors

CURSOR ATTRIBUTES

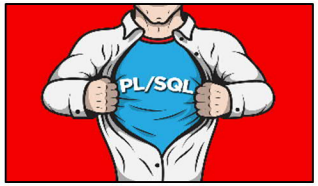
- There are four cursor attributes
  - %FOUND – *Returns true if the fetch returned a row.*
  - %NOTFOUND – *Opposite of %found*
  - %ISOPEN – *Returns true if the cursor is open*
  - %ROWCOUNT – *Returns the number of fetched rows*



Cursors

- **FOR UPDATE CLAUSE**
  - When you update a row, it is locked to the others
  - **for update** clause locks the selected rows
  - **nowait** option will terminate execution if there is a lock
  - Default option is **wait**
  - **for update of** clause locks only the selected tables

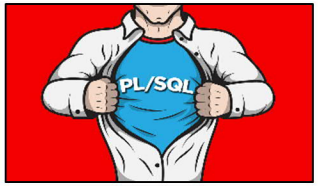
```
cursor cursor_name(parameter_name datatype,…)
   is select_statement
   for update [of column(s)] [nowait | wait n]
```



Cursors

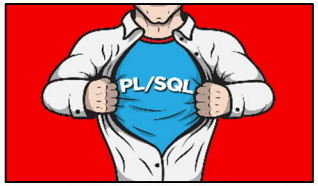
- **REFERENCE CURSORS (REF CURSORS)**
  - Why we need REF CURSORS?
  - Cursors are pointers.
  - You can use a cursor for multiple queries
  - We cannot :
    - ⊗ Assing null values
    - ⊗ Use in table-view create codes
    - ⊗ Store in collections
    - ⊗ Compare
  - How do we use ref cursors?
  - There two types of reference cursors
    - ➢ Strong (restrictive) cursors
    - ➢ Weak (nonrestrictive) cursors.

```
type cursor_type_name is ref cursor [return return_type]
open cursor_variable_name for query;
```



Exceptions

- **What are the Exceptions?**
  - What does compilers do in basic?
  - What are exceptions?
  - How to handle the exceptions?
  - A block has three sections.
    - Declaration section
    - Begin-End Section
    - Exception Section
  - We can handle exceptions in three ways.
    1. Trap
    2. Propagate
    3. Trap & Propagate
  - How to know the exceptions?

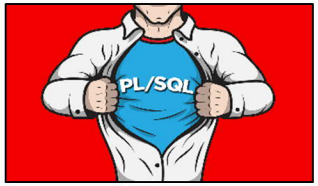


Exceptions

- **Handling Exceptions**

```
declare
    ..............................
begin
    {An exception occurs here}
exception
    when exception_name then
        ..............................
    when others then
        ..............................
end;
```

- There are three types of exceptions :
  - Predefined Oracle Server Errors
  - Nonpredefined Oracle Server Errors
  - User-defined Errors



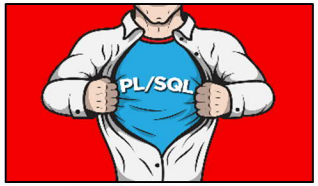
Exceptions

- **Non-predefined Exceptions**
  - Unnamed Exceptions
  - We cannot trap with the error codes
  - We declare exceptions with the error codes

```
exception_name EXCEPTION;
```

```
PRAGMA EXCEPTION_INIT(exception_name, error_code)
```

  - What is PRAGMA ?
  - What does exception_init do?



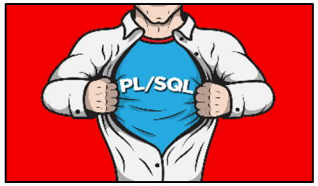
Exceptions

- **User-predefined Exceptions & Raising Exceptions**
  - You need to handle some exceptions about your business.
  - These exceptions are not an error for the database.
  - Define a user-defined exception and raise it.

```
exception_name EXCEPTION;
```

```
RAISE exception_name;
```

  - You can raise any type of exceptions with RAISE statement.

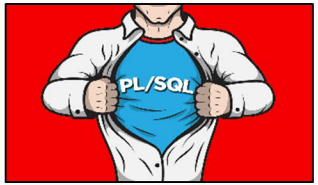


Exceptions

- **RAISE_APPLICATION_ERROR Procedure**
  - Sometimes you want to raise an exception out of the block
  - raise_application_error raises the error to the caller

```
raise_application_error(error_number,error_message[,TRUE|FALSE]);
```

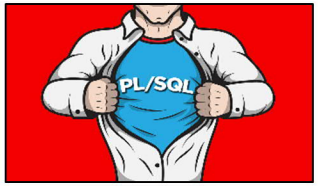
  - What is the error stack?
  - Will stop execution of the application
  - Error number must be between -20000 and -20999
  - Message will be up to 2048 bytes long



Functions & Procedures

What are Functions & Stored Procedures?

- Reusability is important in programming
- What are the differences than anonymous blocks?
  - Stored in the database with names.
  - Compiled only once
  - Can be called by another block or application
  - Can return values
  - Can take parameters

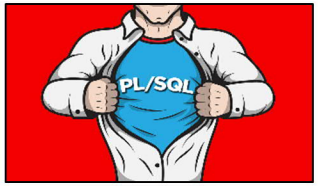


Functions & Procedures

Creating & Using Stored Procedures

- If you need to do the same thing again and again, you use procedures.

```
CREATE [OR REPLACE] PROCEDURE procedure_name
  [(parameter_name [IN | OUT | IN OUT] type [, ...])] {IS | AS}
    ..........................  --declarative section
begin
    ..........................
exception
    ..........................
end;
```

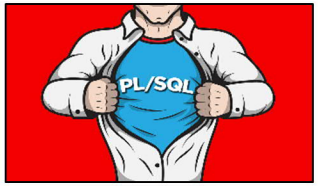


Functions & Procedures

Using IN & OUT Parameters

- Why we pass parameters?
- We can create our procedures and functions with parameters

```
CREATE [OR REPLACE] PROCEDURE procedure_name
  [(parameter_name [IN | OUT | IN OUT] type [, ...])] {IS | AS}
    ..... --declarative section
begin
    .....
exception
    .....
end;
```



Functions & Procedures

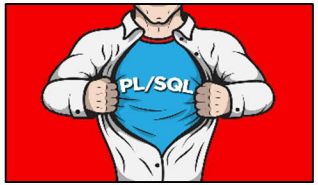
NAMED - MIXED NOTATIONS & DEFAULT OPTION

- We can run the procedures with or without functions
- We do that with the DEFAULT option

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

- Named notation allows us to pass parameter independent from the position.

```
EXECUTE procedure_name(parameter_name => value|expression);
```

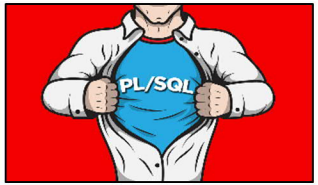


Functions & Procedures

CREATING AND USING FUNCTIONS

- Functions are pretty similar with the procedures
- Functions can get IN and OUT parameters
- Functions must RETURN a value
- Functions are very similar with procedures on creation, except its usage
- Functions can be used within a select statement
- You can assign a function to a variable.

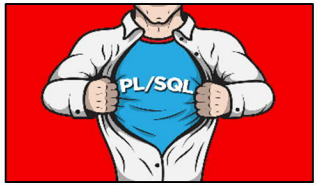
```
CREATE [OR REPLACE] FUNCTION function_name
  [(parameter_name [IN | OUT | IN OUT] type DEFAULT value|expression
  [, ...])] RETURN return_data_type {IS | AS}
```



Functions & Procedures

CREATING AND USING FUNCTIONS

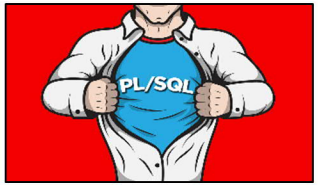
- **Differences and Similarities of Functions & Procedures**
  - Procedures are executed within a begin-end block or with execute command.
  - Functions are used within an SQL Query or assigned to some variable.
  - We can pass IN & OUT parameters to both
  - Procedures does not return a value, but functions return



Functions & Procedures

CREATING AND USING FUNCTIONS

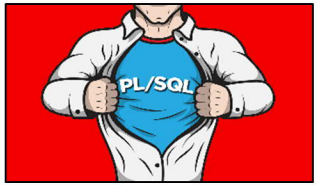
- **The restrictions of using functions in SQL expressions**
  - Must be compiled and stored in the database
  - Your functions should not have an OUT parameter.
  - Must return a valid type of the SQL Data Types
  - Cannot be used in table creation codes
  - Cannot call a function that contains a DML Statement
  - Cannot include COMMIT, ROLLBACK or DDL Statements
  - If the function has a DML operation of the specified table
  - You need to have the related privilege



Functions & Procedures

LOCAL SUBPROGRAMS

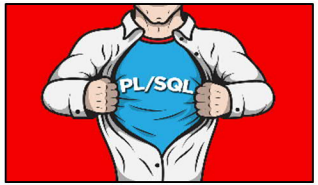
- We create procedures and functions to reduce the code crowd
- They are called as stored procedures & stored functions
- Sometimes it is unnecessary to store the subprograms
- We can create subprograms inside of an anonymous blocks or in another subprogram.
- The benefits of local subprograms :
  - ✓ Reduce code repetition
  - ✓ Improve code readability
  - ✓ Need no more privilege
- The cons of the local variables :
  - ✓ They are accessible only in the blocks they are defined



Functions & Procedures

OVERLOADING THE SUBPROGRAMS

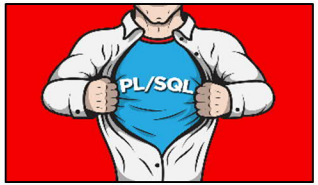
- Overloading means creating more than one subprogram with the same name.
- Overload the subprograms with same name but with different parameters.
- Overloading is pretty useful when using subprograms with packages
- Overloading the Subprograms :
  - ✓ Enables creating two or more subprograms with the same name.
  - ✓ So we can build more flexible subprograms.
  - ✓ We can overload local subprograms and package subprograms. But not standalone subprograms!..
  - ✓ Parameters must be different in data types or orders or numbers.
  - ✓ If parameters are in the same family or subtype, it won't work.
  - ✓ Differentiating only the return type will not be accepted



Functions & Procedures

USING HOST VARIABLES

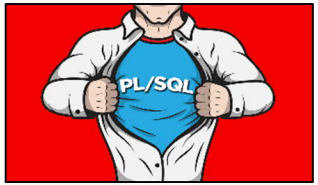
- Overloading means creating more than one subprogram with the same name.
- Overload the subprograms with same name but with different parameters.
- Overloading is pretty useful when using subprograms with packages
- Overloading the Subprograms :
  - ✓ Enables creating two or more subprograms with the same name.
  - ✓ So we can build more flexible subprograms.
  - ✓ We can overload local subprograms and package subprograms. But not standalone subprograms!..
  - ✓ Parameters must be different in data types or orders or numbers.
  - ✓ If parameters are in the same family or subtype, it won't work.
  - ✓ Differentiating only the return type will not be accepted



Functions & Procedures

HANDLING THE EXCEPTIONS IN SUBPROGRAMS

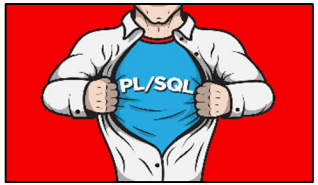
- Why to handle the exceptions in subprograms?
- You can have an exception section in all begin-end blocks.
- When an exception is raised, the control is passed to the exception section.
- What happens if we don't handle the exceptions?
- We may have exceptions in functions & procedures, too.
- In subprograms, the unhandled exception is raised to the caller.
- The control must be on you.
- You should handle all the possible exceptions.



Functions & Procedures

REGULAR & PIPELINED TABLE FUNCTIONS

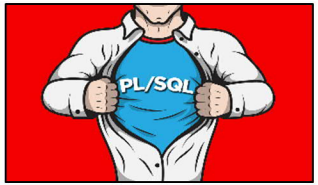
- What are table functions?
- Table functions return a table of varray or nested tables
- Regular table functions returns after completing the whole data
- Pipelined functions return each row one by one
- Which one to choose?



Packages

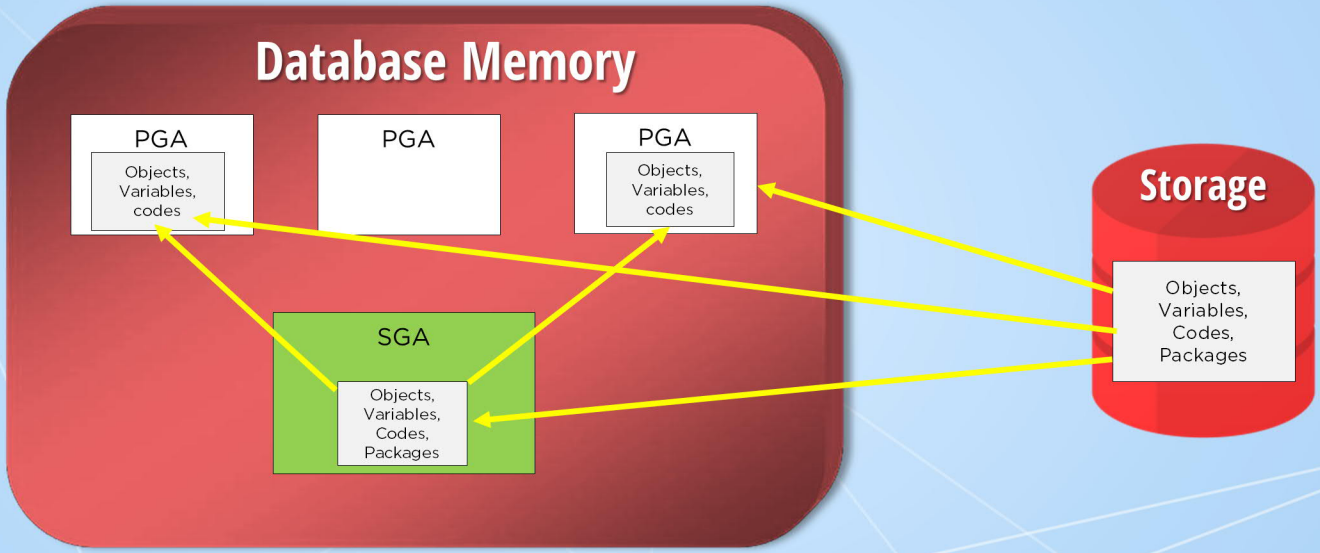
WHAT IS A PACKAGE?

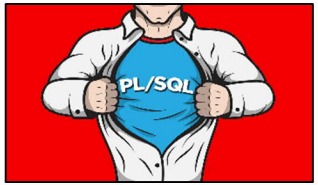
- Most of the times, our objects work together.
- There will be an object crowd in a real work in time
- Packages groups subprograms, types, variables etc in one container.
- There is one more reason for using packages



Packages

DIFFERENCE OF A PACKAGE USAGE IN MEMORY

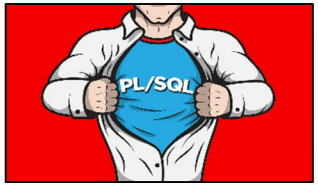




Packages

ADVANTAGES OF USING PACKAGES

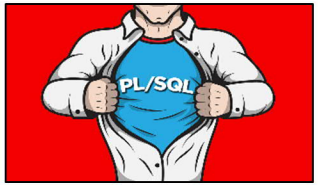
- Modularity
- Easy Maintenance
- Encapsulation & Security
- Performance
- Functionality
- Overloading



Packages

CREATING & USING PACKAGES

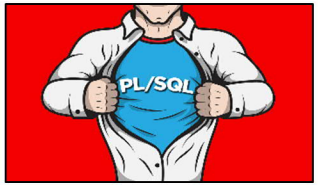
- To use a package, we have two main reasons :
  - ✓ Logically grouping the objects
  - ✓ The performance gain
- With logically grouping, we can decrease code complexity and crowd.
- A package consists of two parts.
  - ➢ Package Specification (spec)
  - ➢ Package Body (body)
- Body and Spec must have the same name



Packages

VISIBILITY OF PACKAGE OBJECTS

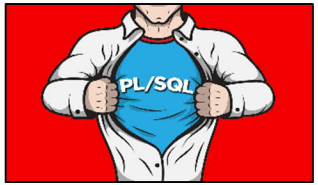
- An object is visible to the others only if it is declared in the package spec.
- Where can we declare the variables in packages?
  - ✓ Inside of the package spec
  - ✓ Inside of the package body



Packages

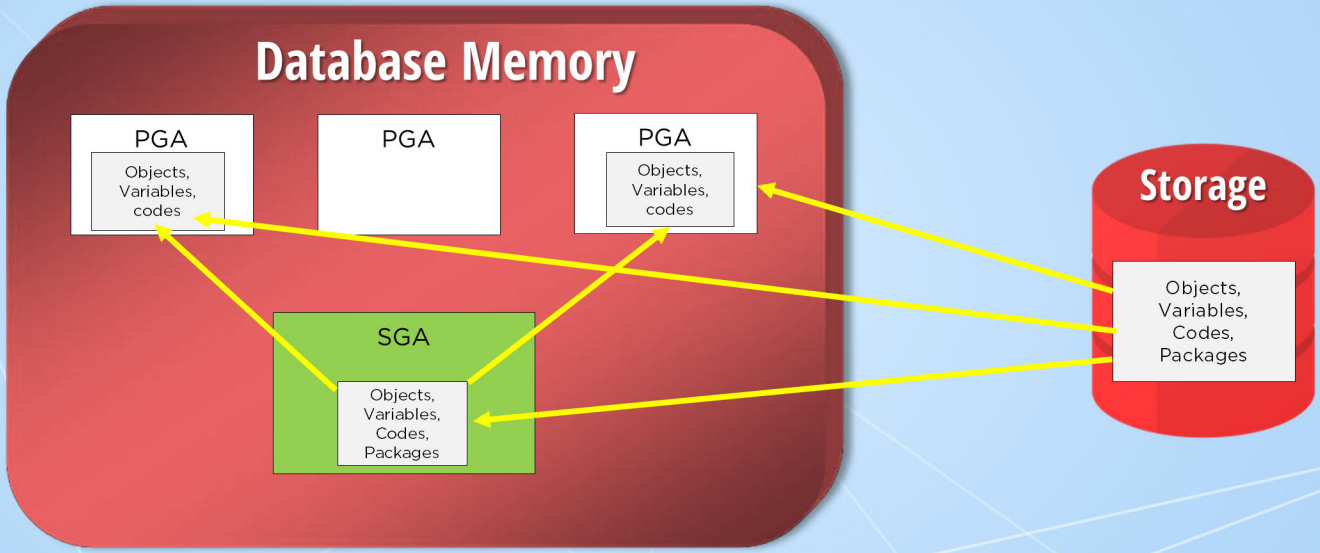
PACKAGE INITIALIZATION

- A package is loaded into the memory on the first call.
- Uninitialized variables are null by default.
- Oracle let's us to initialize the variables with one more way.
- The initialized variables will be overridden with that way.
- We can also do some business here.
- We do that as the last begin-end block.

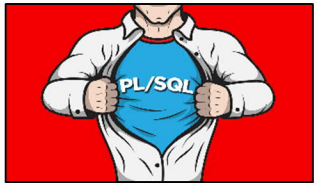


Packages

PERSISTENT STATE OF PACKAGES



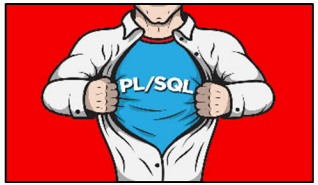
 |



Packages

PERSISTENT STATE OF PACKAGES

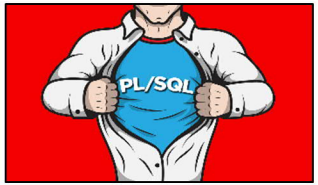
- A package is loaded into the memory at the first reference.
- Variables and objects are stored in your PGA.
- These variables are persistent for your session.
- These variables are unique for your session.
- Subprograms of the packages are stored in the SGA.
- We can change the persistent state of variables.
- PRAGMA SERIALLY_REUSABLE
- Serially reusable packages cannot be accessed from :
  - Triggers
  - Subprograms called from SQL Statements.



Packages

HOW TO FIND THE PACKAGES?

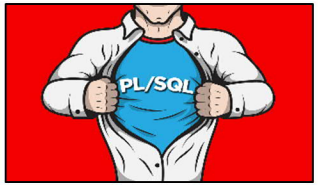
- We can use SQL Developer Tool
- We can use Data Dictionary Views
- The Data Dictionary Views that you can find some information about the packages are :
    - USER_SOURCE / ALL_SOURCE / DBA_SOURCE
    - USER_OBJECTS / ALL_OBJECTS / DBA_OBJECTS



Triggers

WHAT ARE TRIGGERS & TRIGGER TYPES?

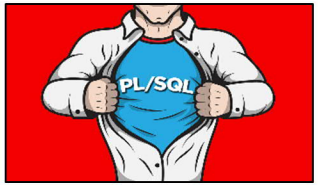
- Triggers are PL/SQL blocks that executed before or after or instead of a specific event.
- Triggers are executed automatically by the database server.
- Triggers are defined on tables, views, schemas, databases.
- Triggers are fired when one of these occurs :
  - When a DML (insert, update, delete) occurs
  - When a DDL (create, alter, drop) occurs
  - When some database operations occurs (logon,startup,servererror,..)
- These are called as database triggers
- Application triggers are related with some applications like Oracle Forms,..



Triggers

WHAT ARE TRIGGERS & TRIGGER TYPES?

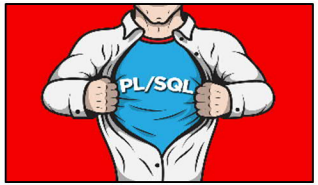
- Why do we use Triggers?
  - ✓ Security
  - ✓ Auditing
  - ✓ Data Integrity
  - ✓ Table Logging
  - ✓ Event Logging
  - ✓ Derived Data
- There are three types of triggers :
  - ➢ DML Triggers
  - ➢ Compound Triggers
  - ➢ Non-DML Triggers



Triggers

UNDERSTANDING DML TRIGGERS

- What are DML Triggers?
- DML Triggers are PL/SQL blocks running when the specified event occurs.
- We use DML Triggers for duplications, log table maintenance, security, etc.
- Let's check out the Trigger Creation Schema

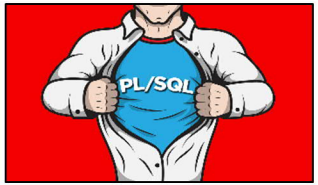


Triggers

UNDERSTANDING DML TRIGGERS

- Trigger Creation Schema

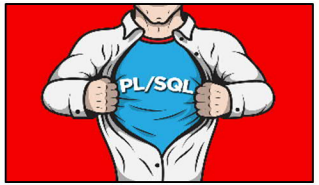
```
CREATE [OR REPLACE] TRIGGER trigger_name
 Timing = BEFORE | AFTER | INSTEAD OF
 Event  = INSERT | UPDATE | DELETE | UPDATE OF column_list
 ON object_name
 [ REFERENCING OLD AS old NEW AS new]
 [ FOR EACH ROW ]
 [ WHEN (Condition) ]
 [ DECLARE variables,types,etc]
BEGIN
   trigger_body
 [ EXCEPTION ]
END;
```



Triggers

SPECIFYING THE TIMING OF TRIGGERS

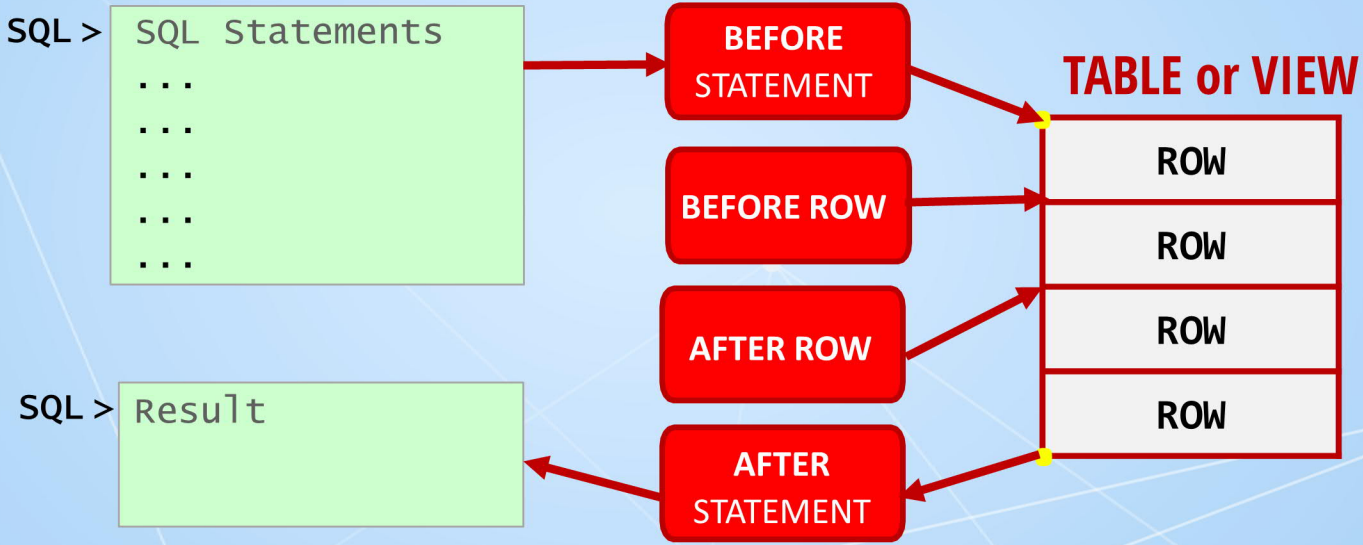
- There are three options to specify the timing
  - BEFORE
    - ✓ We can allow or reject the specified action
    - ✓ We can specify default values for the columns
    - ✓ We can validate complex business rules
  - AFTER
    - ✓ Make some after checks
    - ✓ Duplicate tables or add Log records
  - INSTEAD OF
- There can be multiple triggers with the same timing points on a table or view

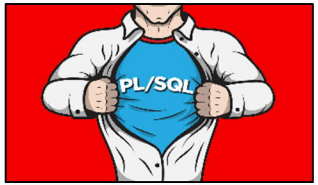


Triggers

STATEMENT & ROW LEVEL TRIGGERS

- Sample schema about when the DML Triggers are executed



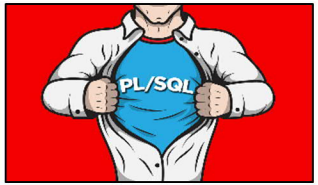


Triggers

:NEW & :OLD QUALIFIERS IN TRIGGERS

- :NEW and :OLD Qualifiers are used in row level triggers
- :old.column_name returns the old value of the column
  :new.colum_name returns the new value of the column
- Old and new values of the columns are stored in two different records implicitly by the Oracle server.

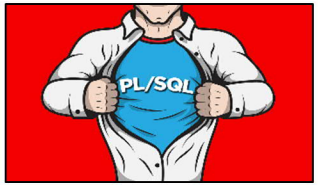
Data Operations	OLD Value	NEW Value
INSERT	NULL	Inserted value
UPDATE	Value before update	Value after update
DELETE	Value before delete	NULL



Triggers

:NEW & :OLD QUALIFIERS IN TRIGGERS

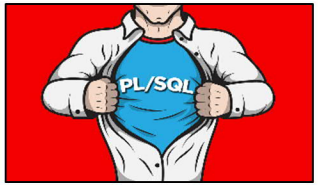
- We use old and new qualifiers to reach the columns.
- We can use old and new qualifiers in an SQL query in your trigger, or in a PL/SQL Statement
- Row level triggers may decrease the performance of your code if so many inserts, updates or deletes occurs
- Colon prefix before the old and new qualifiers are not used in the when condition
- Let's use them..



Triggers

INSTEAD OF TRIGGERS

- Simple Views enable DMLs, but Complex views do not.
- Instead of triggers are used to apply some DML statements on un-updatable views
- Some important things about the instead of triggers :
  - Instead of triggers are used only with the views.
  - Generally used for the complex views.
  - If your view has a check option, it won't be enforced when you use the instead of triggers
  - Before and after timing options are not valid for instead of triggers.



Triggers

EXPLORING & MANAGING THE TRIGGERS

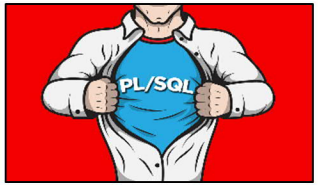
- There are two data dictionary views for the triggers : USER_OBJECTS & USER_TRIGGERS

```
ALTER TRIGGER trigger_name ENABLE | DISABLE;
```

```
ALTER TABLE base_object_name [ENABLE | DISABLE] ALL TRIGGERS;
```

```
ALTER TRIGGER trigger_name COMPILE;
```

```
DROP TRIGGER trigger_name;
```



Triggers

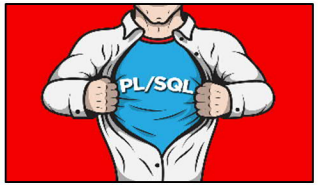
COMPOUND TRIGGERS

- Compound trigger is a single trigger that allows us to specify actions for each DML trigger types.
- So they can share the variables, types etc among each other.

```
CREATE OR REPLACE TRIGGER schema.trigger
FOR dml_event_cluase ON schema.table
COMPOUND TRIGGER
-- Initialization Section
   --Declaration area
   --Subprograms
BEFORE STATEMENT IS
  ...
END BEFORE STATEMENT;
AFTER STATEMENT IS
  ...
END AFTER STATEMENT;
BEFORE EACH ROW IS
  ...
END BEFORE EACH ROW;
AFTER EACH ROW IS
  ...
END AFTER EACH ROW;
```

DECLARATION

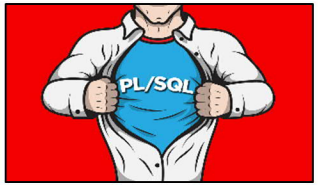
OPTIONAL SECTIONS



Triggers

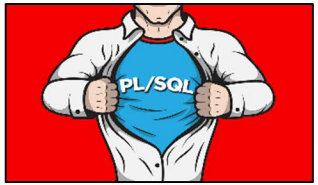
COMPOUND TRIGGERS

- Why we use the compound triggers?
  - ✓ Taking actions for various timing points by sharing the common data
  - ✓ Making inserts to some other tables faster with bulk inserts.
  - ✓ Avoiding mutating table errors.
- Compound Trigger Restrictions :
  - ➢ A compound trigger must be a DML trigger defined on a table or view.
  - ➢ A compound trigger body must be a compound trigger block.
  - ➢ A compound trigger body cannot have an initialization block.
  - ➢ :OLD and :NEW cannot be used in the declaration or before or after statements.
  - ➢ The firing order of compound triggers is not guaranteed if you don't use the follows clause.



Triggers

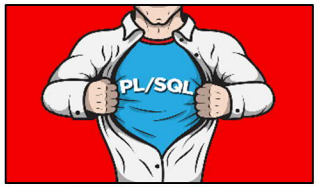
```
CREATE [OR REPLACE] TRIGGER trigger_name
FOR INSERT | UPDATE | DELETE | UPDATE OF column_list
 ON object_name
 [ REFERENCING OLD AS old NEW AS new]
 [ WHEN (Condition) ]
COMPOUND TRIGGER
 [variables,types,etc]
    BEFORE STATEMENT IS
       PL/SQL Block…
     [ EXCEPTION ]
    END BEFORE STATEMENT;
    BEFORE EACH ROW IS
       PL/SQL Block…
     [ EXCEPTION ]
    END BEFORE EACH ROW;
    AFTER EACH ROW IS
       PL/SQL Block…
     [ EXCEPTION ]
    END AFTER EACH ROW;
    AFTER STATEMENT IS
       PL/SQL Block…
     [ EXCEPTION ]
    END AFTER STATEMENT;
END;
```



Triggers

MUTATING TABLE ERRORS

- What are mutating table errors?
- Trigger Restrictions on Mutating Tables
  - A Mutating table is :
    - A table that being modified.
    - A table that might be updated with the DELETE CASCADE.
- Row Level triggers cannot query or modify a mutating table.
- This restriction prevents inconsistent data changes.
- Views being modified by the instead of triggers are not considered as mutating.
- We can handle mutating table errors with a couple ways:
  - ✓ Store related data in another table.
  - ✓ Store related data in a Package
  - ✓ Use Compound Triggers



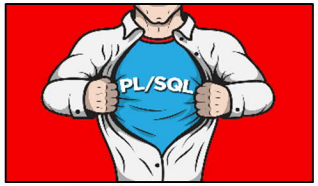
Using Dynamic SQL, PL&SQL in PL/SQL

Introduction to Dynamic SQL & Dynamic PL/SQL:

- Oracle supports 2 types of SQL:

Static SQL **DYNAMIC SQL**

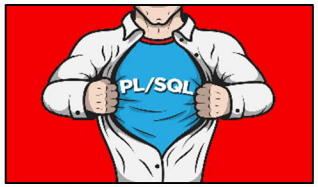
- SQL Execution Stages:
  - **PARSE**
  - **BIND**
  - **EXECUTE**
  - **FETCH**
- Static SQL:
  - Written in advance.
  - Parsed at compile time.



Using Dynamic SQL, PL&SQL in PL/SQL

Introduction to Dynamic SQL & Dynamic PL/SQL:

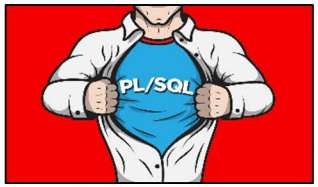
- We may need to generate some queries at runtime
- The Dynamic SQL is a SQL statement constructed as strings and executed dynamically at **runtime**.
- Why do we need the dynamic SQL?
  - We may not know the full-text a statement to be executed in advance.
  - We may want to execute DDL statements or SQL statements that are not supported in static SQL statements.
  - We may need dynamic PL/SQL blocks that will work different in different situations
- The Dynamic SQL may lead to some problems such as errors, privilege problems, SQL injections etc.
- Static SQL should be preferred over the Dynamic SQL, when possible.
- The Dynamic SQLs are usually more complicated, harder to debug, slower and harder to maintain.



Using Dynamic SQL, PL&SQL in PL/SQL

Introduction to Dynamic SQL & Dynamic PL/SQL:

- Advantages of Dynamic SQL :
  - Constructed as string at runtime.
  - Bind variables can be used.
  - Allows us to use the DDL, DCL or Session-Control statements in PL/SQL blocks.
- How to generate Dynamic SQL?
  - **Native Dynamic SQL Statements**
  - **DBMS_SQL Package**
- We can also generate dynamic PL/SQL blocks.
- The SQL Injection is a technique for placing malicious code into your Dynamic SQL statements.
- If you can use static SQL instead of Dynamic SQL, it will be a better option.



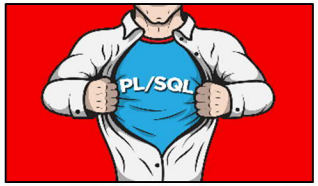
Using Dynamic SQL, PL&SQL in PL/SQL

Native Dynamic SQL:

- Native Dynamic SQL statements are used in two ways :
  - EXECUTE IMMEDIATE Command
  - OPEN-FOR, FETCH and CLOSE Statements
- Both ways of Native Dynamic SQL statements handle IN, IN OUT and OUT bind variables that are bound by position, not by name.

```
EXECUTE IMMEDIATE Dynamic_SQL_String
 [[BULK COLLECT] INTO {variable [,{variable]... | record}]
 [USING [IN|OUT|IN OUT] bind_argument
      [,[IN|OUT|IN OUT] bind_argument]...];
```

- The Dynamic SQL string should be without a terminator if it is a SQL statement, or with a terminator if it is a PL/SQL block.
- We can use numeric, character and string literals as bind arguments, but we cannot use NULL.



Using Dynamic SQL, PL&SQL in PL/SQL

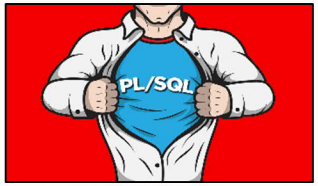
Native Dynamic SQL (continued):

- We can use the EXECUTE IMMEDIATE statements in 3 ways :

```
EXECUTE IMMEDIATE Dynamic_SQL_String;
```

```
EXECUTE IMMEDIATE Dynamic_SQL_String
 [USING [IN|OUT|IN OUT] bind_argument
      [,[IN|OUT|IN OUT] bind_argument] … ];
```

```
EXECUTE IMMEDIATE Dynamic_SQL_String
 [[BULK COLLECT] INTO {variable [,{variable] … | record}]
 [USING [IN|OUT|IN OUT] bind_argument
      [,[IN|OUT|IN OUT] bind_argument] … ];
```



Using Dynamic SQL, PL&SQL in PL/SQL

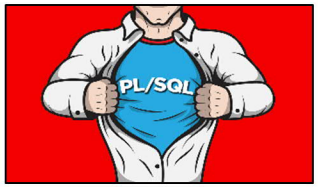
Native Dynamic SQL (continued):

- The use of the OPEN-FOR, FETCH and CLOSE Statements:

```
OPEN cursor FOR Dynamic_SQL_String
 [USING bind_argument [,bind_argument] ... ];
```

```
FETCH cursor INTO variable [,variable] …[,record];
```

```
CLOSE cursor;
```



Using Dynamic SQL, PL&SQL in PL/SQL

Native Dynamic PL/SQL:

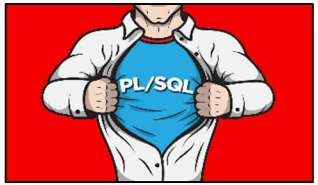
- The use of the Dynamic PL/SQL:

```
EXECUTE IMMEDIATE Dynamic_PL/SQL_String(Must include BEGIN-END)
 [USING IN|IN OUT|OUT bind_argument
      [,IN|IN OUT|OUT bind_argument] … ];
```

Helpful Tips

Tips you should know when working with the dynamic PL/SQL;

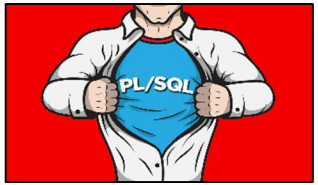
- The dynamic string must be a valid PL/SQL block.
- The string must end with a semicolon.
- The dynamic PL/SQL block can only access global objects.
- The errors raised within a dynamic PL/SQL block can be trapped and handled by the surrounding block.



Using Dynamic SQL, PL&SQL in PL/SQL

The DBMS_SQL Package

- Native Dynamic SQL should be your first choice over DBMS_SQL when possible.
- Native Dynamic SQL is similar to static SQL and much easier to code than the DBMS_SQL.
- Native Dynamic SQL is better than using the DBMS_SQL package in performance aspect.
- Native Dynamic SQL supports more object types than the DBMS_SQL package.
- The DBMS_SQL Package doesn't have the cursor attributes like **%found**, **%rowcount**, etc.
- Why to use the DBMS_SQL Package?
  - You may encounter some DBMS_SQL codes in your job.
  - When the Dynamic SQL Statement has an unknown number of bind variables or columns until runtime (The Method 4).
  - When executing the same SQL Statement multiple times with different bind values.



Using Dynamic SQL, PL&SQL in PL/SQL

The DBMS_SQL Package

- **How to use the DBMS_SQL Package?**
  - The subprograms of the DBMS_SQL Package are:

Function **OPEN_CURSOR():** Opens a new cursor and returns the cursor ID.

Procedure **PARSE(cursor_id, statement, edition):** Parses the dynamic SQL statement.

Function **EXECUTE(cursor_id) :** Executes the statement and returns the number of rows processed.

Function **FETCH_ROWS(cursor_id) :** Retrieves the next row.

Procedure **CLOSE_CURSOR(cursor_id):** Closes the given cursor.

Procedure **DEFINE_COLUMN(cursor_id, position, column) :** Adds the column to the column list.

Procedure **BIND_VARIABLE(cursor_id, name, value):** Binds values to the query.

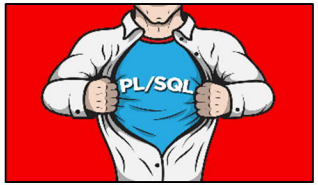
Procedure **COLUMN_VALUE(cursor_id, position, value) :** Retrieves the selected column value.

- There are many more subprograms in the DBMS_SQL package…

Oracle-Supplied Packages

What are Oracle-Supplied Packages?

- There are so many built-in packages to solve frequent problems or the operations that we can't do with the existing PL/SQL codes.
- There are over 1000 built-in packages and increasing.
- Some packages are not installed by default. The DBAs need to install them explicitly when there is a need for that.
- Some very useful packages :
  - **STANDARD Package** Includes so many subprograms, types, exceptions, etc. that we use frequently while coding in PL/SQL.
  - **DBMS_OUTPUT Package** Used to debug subprograms and read-write text data.
  - **UTL_FILE Package** Performs file operations.
  - **UTL_MAIL Package** Performs mailing operations.
  - And others like **DBMS_SQL, DBMS_ALERT**, **DBMS_LOB**, …



Oracle-Supplied Packages

The DBMS_Output Package

- The DBMS_OUTPUT package provides debugging and buffering of text data.
- The procedures of the DBMS_OUTPUT package are:

Procedure **PUT (item IN VARCHAR2):** Appends text to the current line of the buffer.

Procedure **PUT_LINE (item IN VARCHAR2):** In addition to what the PUT procedure does, this adds an additional end-of-line character.

Procedure **NEW_LINE:** Adds an end-of-line(new line) character.

- We can pass maximum 32767 bytes of data in one put or put_line procedure call.
- The SET SERVEROUTPUT ON command displays the buffered text automatically on the screen.
- The output will be printed after the entire code finishes executing.

Procedure **GET_LINE(line OUT VARCHAR2, status OUT INTEGER) :** Gets the current line of text from the buffer starting from the first line.

Procedure **GET_LINES(lines OUT CHARARR, num_of_lines IN OUT INTEGER):** Gets the current line of text from the buffer starting from the first line.

- We cannot get a line if it is not terminated with a new_line character.

Procedure **DISABLE :** Disables the calls to PUT, PUT_LINE, NEW_LINE, GET_LINE and GET_LINES procedures.

Procedure **ENABLE(buffer_size IN INTEGER DEFAULT 20000):** Enables the package for operations and assigns a buffer with the given size.

Oracle-Supplied Packages

The UTL_FILE Package

- It reads and writes to any operating system files that are accessible to the database server.
- It provides file access on both the server and client sides.
- The `UTL_FILE` operates on:
  - The paths(directories) specified in the `UTL_FILE_DIR` parameter:

```
UTL_FILE_DIR = directory
```

Windows

```
UTL_FILE_DIR = C:\My_files
```

Linux

```
UTL_FILE_DIR = /users/test_area
```

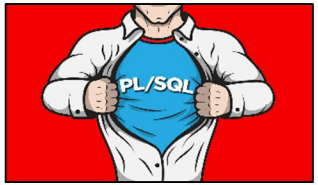
  - Database directory aliases created for specific paths(directories)

```
CREATE OR REPLACE DIRECTORY test_dir AS 'C:\my_files';
```

    - The database does not check the path for existence.
    - Directory names must be written in uppercase when using the `UTL_FILE` subprograms.
    - We can see the existing directories by looking at the **`DBA_DIRECTORIES`** and **`ALL_DIRECTORIES`** views.
    - Users can have different privileges on the same directory.

```
GRANT WRITE ON test_dir TO my_user;
```

```
GRANT READ ON test_dir TO hr;
```



Oracle-Supplied Packages

The UTL_FILE Package

➢ The UTL_FILE has the following subprograms:

- **FOPEN Function:** Opens a file to read, write or append.

```
FOPEN (   location      IN VARCHAR2,
          file_name     IN VARCHAR2,
          open_mode     IN VARCHAR2,
          max_linesize  IN BINARY_INTEGER DEFAULT NULL)
RETURN file_type;
```

```
TYPE file_type IS RECORD(id          BINARY_INTEGER,
                         data_type   BINARY_INTEGER,
                         byte_mode   BOOLEAN);
```

Open Modes

- r ➔ Read Text
- w ➔ Write Text
- a ➔ Append Text
- rb ➔ Read Binary
- wb ➔ Write Binary
- ab ➔ Append Binary

- **IS_OPEN Function:** Returns true if the file is already open.

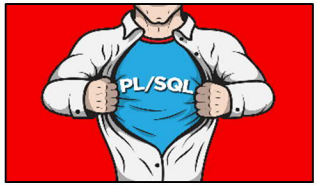
```
IS_OPEN (file IN FILE_TYPE) RETURN BOOLEAN;
```

- **FCLOSE Procedure:** Closes an open file with the given file handle.

```
FCLOSE (file IN OUT FILE_TYPE);
```

- **FCLOSE_ALL Procedure:** Closes all the open file handles created in that session.

```
FCLOSE_ALL;
```



Oracle-Supplied Packages

The UTL_FILE Package

- **GET_LINE Procedure:**

  Reads text from an open file

```
GET_LINE(file   IN FILE_TYPE,
         buffer OUT VARCHAR2,
         len    IN BINARY_INTEGER DEFAULT NULL);
```

- **PUT Procedure:**

  Writes a text to the file without a newline character.

```
PUT     (file   IN FILE_TYPE,
         buffer IN VARCHAR2);
```

- **NEW_LINE Procedure:**

  Writes one ore more line terminators to the file.

```
NEW_LINE (file  IN FILE_TYPE,
          lines IN BINARY_INTEGER := 1);
```

- **PUT_LINE Procedure:**

  Writes a text to the file with a newline character.

```
PUT_LINE(file      IN FILE_TYPE,
         buffer    IN VARCHAR2,
         autoflush IN BOOLEAN DEFAULT FALSE);
```

- **FFLUSH Procedure:**

  Physically writes the buffered data to disks.

```
FFLUSH   (file IN FILE_TYPE);
```

- **PUTF Procedure:**

  Provides formatted put procedure.

```
PUTF    (file      IN FILE_TYPE,
         format    IN VARCHAR2,
         arguments IN VARCHAR2 DEFAULT NULL);
```

Oracle-Supplied Packages

The UTL_FILE Package

- **FCOPY Procedure:** Copies a portion or the complete file to a newly created file.

```
FCOPY     ( src_location       IN VARCHAR2,
            src_filename       IN VARCHAR2,
            dest_location      IN VARCHAR2,
            dest_filename      IN VARCHAR2,
            start_line         IN BINARY_INTEGER DEFAULT 1,
            end_line           IN BINARY_INTEGER DEFAULT NULL);
```

- **FRENAME Procedure:** Can rename and move a file to another destination.

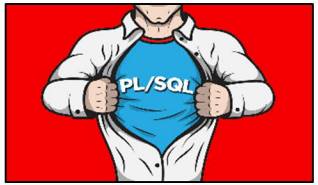
```
FRENAME   ( src_location       IN VARCHAR2,
            src_filename       IN VARCHAR2,
            dest_location      IN VARCHAR2,
            dest_filename      IN VARCHAR2,
            overwrite          IN BOOLEAN DEFAULT FALSE);
```

- **FREMOVE Procedure:** Removes the file from the disk.

```
FREMOVE   ( src_location       IN VARCHAR2,
            src_filename       IN VARCHAR2);
```

- **FGETATTR Procedure:** Retrieves the existence, length and file-system block size information.

```
FGETATTR  ( location           IN VARCHAR2,
            filename           IN VARCHAR2,
            fexists            OUT BOOLEAN,
            file_length        OUT NUMBER,
            block_size         OUT BINARY_INTEGER);
```



Oracle-Supplied Packages

The UTL_FILE Package

- Function **FOPEN_NCHAR:** Opens a file in Unicode for input or output.
- Function **FGETPOS:** Returns the current relative offset position within a file in bytes.
- Procedure **FSEEK:** Adjusts the file pointer forward or backward within the file.
- Procedure **GET_LINE_NCHAR:** Retrieves a line from the file in Unicode form.
- Procedure **GET_RAW:** Reads a raw string value from a file.
- Procedure **PUT_LINE_NCHAR:** Writes a Unicode string to a file with the newline character.
- Procedure **PUT_NCHAR:** Writes a Unicode string to a file without the newline character.
- Procedure **PUTF_NCHAR:** Provides formatted put_nchar procedure.
- Procedure **PUT_RAW:** Writes RAW data value to a file.

Oracle-Supplied Packages

The UTL_MAIL Package

- The UTL_MAIL package is used to send emails that include commonly used email features.
- There are some other packages that can be used to send emails such as **UTL_TCP**, **UTL_SMTP**, **APEX_MAIL**…
- The UTL_MAIL package is very easy to use.
- The UTL_MAIL package is not installed by default. You need to install it.
- Here are the steps to install & configure the UTL_MAIL package.

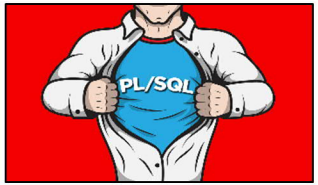
```
@?\rdbms\admin\utl_mail.sql
@?\rdbms\admin\prvtmail.plb
```

```
@ORACLE_HOME\rdbms\admin\utlmail.sql
@ORACLE_HOME\rdbms\admin\prvtmail.plb
```

```
@C:\OracleApp\WINDOWS.X64_193000_db_home\rdbms\admin\utl_mail.sql
@C:\OracleApp\WINDOWS.X64_193000_db_home\rdbms\admin\prvtmail.plb
```

```
SELECT * FROM v$parameter WHERE name = 'smtp_out_server';
```

```
ALTER SYSTEM SET smtp_out_server = 'your_smtp_server';
```



Oracle-Supplied Packages

The UTL_MAIL Package

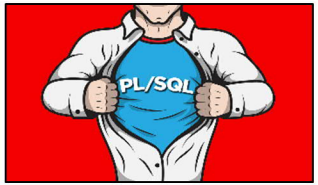
- **Configuring Network Security:**

  Creating **ACL**(**A**ccess **C**ontrol **L**ist) and granting privilege to the user.

```
BEGIN
  DBMS_NETWORK_ACL_ADMIN.CREATE_ACL (
    acl         => 'mail-server.xml',
    description => 'Granting privs to required users for UTL_SMTP.xml',
    principal   => 'YOUR_USER',
    is_grant    =>  TRUE,
    privilege   => 'connect');

  DBMS_NETWORK_ACL_ADMIN.ADD_PRIVILEGE (
    acl         => 'mail-server.xml',
    principal   => 'YOUR_USER',
    is_grant    =>  TRUE,
    privilege   => 'resolve');

  DBMS_NETWORK_ACL_ADMIN.ASSIGN_ACL (
    acl         => 'mail-server.xml',
    host        => 'the_smtp_server');
END;
COMMIT;
```



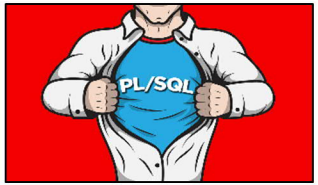
Oracle-Supplied Packages

The UTL_MAIL Package

SEND Procedure

Sends messages to the related SMTP server for forwarding to the recipients.

```
SEND (sender     IN VARCHAR2     CHARACTER SET ANY_CS,
      recipients IN VARCHAR2     CHARACTER SET ANY_CS,
      cc         IN VARCHAR2     CHARACTER SET ANY_CS DEFAULT NULL,
      bcc        IN VARCHAR2     CHARACTER SET ANY_CS DEFAULT NULL,
      subject    IN VARCHAR2     CHARACTER SET ANY_CS DEFAULT NULL,
      message    IN VARCHAR2     CHARACTER SET ANY_CS DEFAULT NULL,
      mime_type  IN VARCHAR2     CHARACTER SET ANY_CS DEFAULT 'text/plain;charset=us-ascii',
      priority   IN PLS_INTEGER  DEFAULT 3,
      replyto    IN VARCHAR2     CHARACTER SET ANY_CS DEFAULT NULL);
```



Oracle-Supplied Packages

The UTL_MAIL Package

SEND_ATTACH_RAW Procedure

Sends emails with a **binary** attachment.

```
SEND_ATTACH_RAW( sender         IN VARCHAR2 CHARACTER SET ANY_CS,
                 recipients     IN VARCHAR2 CHARACTER SET ANY_CS,
                 cc             IN VARCHAR2 CHARACTER SET ANY_CS DEFAULT NULL,
                 bcc            IN VARCHAR2 CHARACTER SET ANY_CS DEFAULT NULL,
                 subject        IN VARCHAR2 CHARACTER SET ANY_CS DEFAULT NULL,
                 message        IN VARCHAR2 CHARACTER SET ANY_CS DEFAULT NULL,
                 mime_type      IN VARCHAR2 CHARACTER SET ANY_CS DEFAULT 'text/plain; charset=us-ascii',
                 priority       IN PLS_INTEGER DEFAULT 3,
                 attachment     IN RAW,
                 att_inline     IN BOOLEAN  DEFAULT TRUE,
                 att_mime_type IN VARCHAR2 CHARACTER SET ANY_CS DEFAULT 'application/octet',
                 att_filename   IN VARCHAR2 CHARACTER SET ANY_CS DEFAULT NULL,
                 replyto        IN VARCHAR2 CHARACTER SET ANY_CS DEFAULT NULL);
```

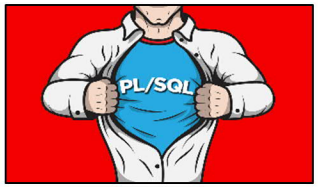
Oracle-Supplied Packages

The UTL_MAIL Package

SEND_ATTACH_VARCHAR2 Procedure

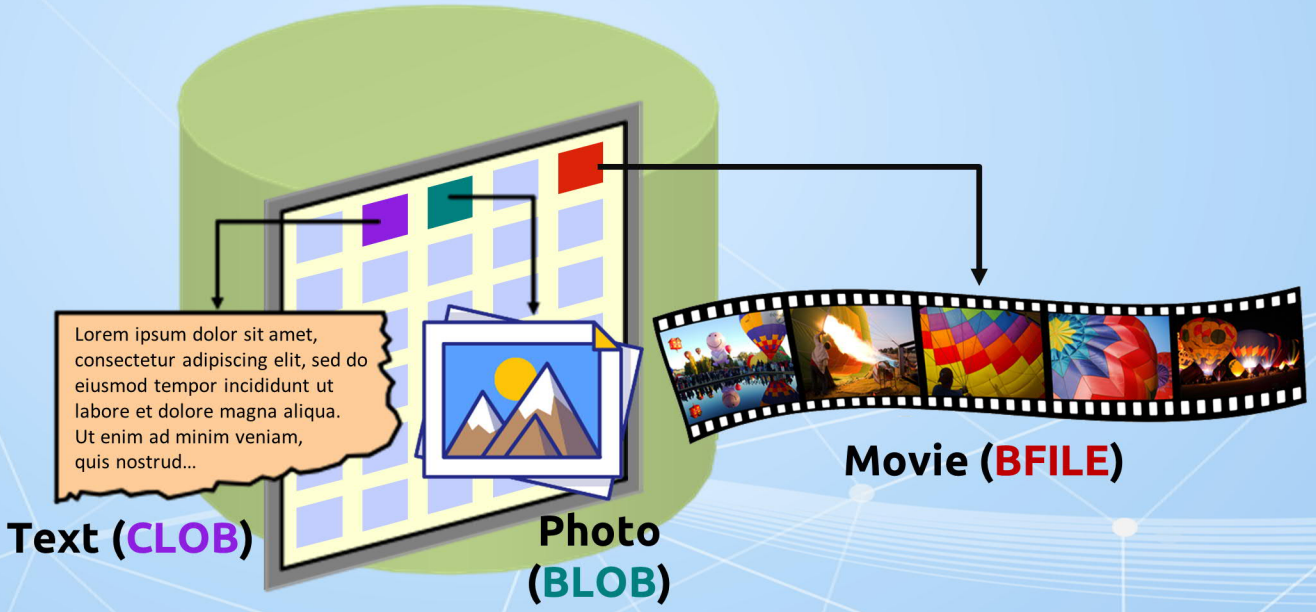
Sends emails with a **binary** attachment.

```
SEND_ATTACH_VARCHAR2(  sender        IN VARCHAR2 CHARACTER SET ANY_CS,
                       recipients    IN VARCHAR2 CHARACTER SET ANY_CS,
                       cc            IN VARCHAR2 CHARACTER SET ANY_CS DEFAULT NULL,
                       bcc           IN VARCHAR2 CHARACTER SET ANY_CS DEFAULT NULL,
                       subject       IN VARCHAR2 CHARACTER SET ANY_CS DEFAULT NULL,
                       message       IN VARCHAR2 CHARACTER SET ANY_CS DEFAULT NULL,
                       mime_type     IN VARCHAR2 CHARACTER SET ANY_CS
                                                 DEFAULT 'text/plain; charset=us-ascii',
                       priority      IN PLS_INTEGER DEFAULT 3,
                       attachment    IN VARCHAR2 CHARACTER SET ANY_CS,
                       att_inline    IN BOOLEAN  DEFAULT TRUE,
                       att_mime_type IN VARCHAR2 CHARACTER SET ANY_CS
                                                 DEFAULT 'text/plain; charset=us-ascii',
                       att_filename  IN VARCHAR2 CHARACTER SET ANY_CS DEFAULT NULL,
                       replyto       IN VARCHAR2 CHARACTER SET ANY_CS DEFAULT NULL);
```

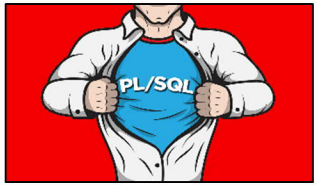



Oracle LARGE Objects (LOBs)

- LOBs are used to store large amounts of data.



 |



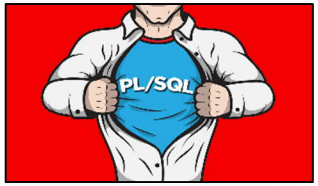
Oracle LARGE Objects (LOBs)

LOB Advantages:

- LOBs an hold large amounts of data.(8TB to 128TB.)
- There can be multiple columns that hold large objects in a single table.
- LOBs support random access to any data which makes it faster.
- LOBs are easier to maintain and can be optimized.
- LOBs can be attributes of an object type.
- LOBs allow us to access data stored outside the database or within the database.

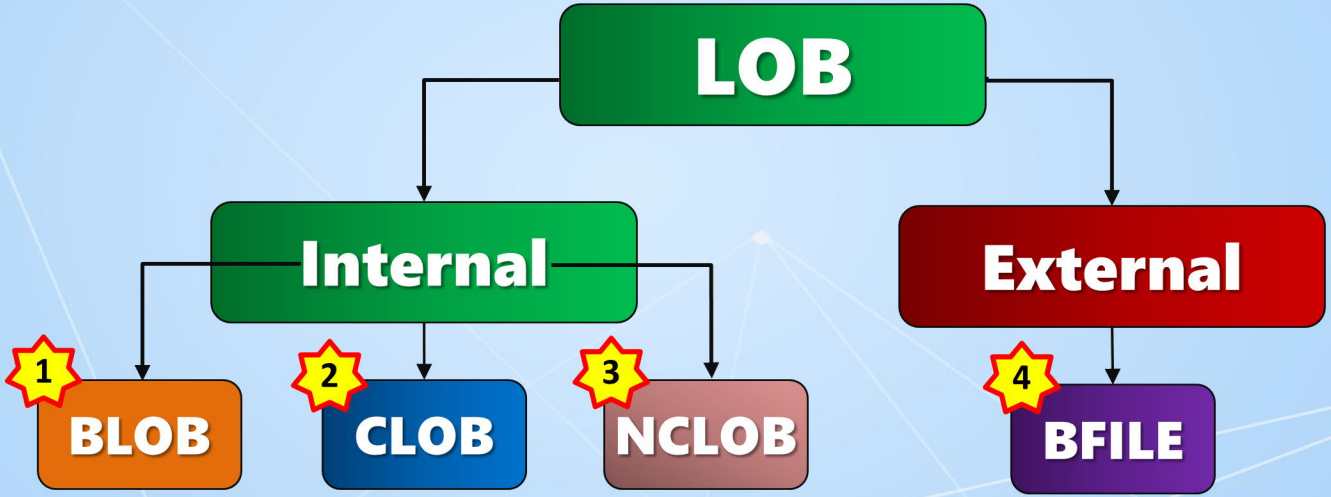
LOB Restrictions:

- ⚠ A LOB column cannot be a primary key of a table.
- ⚠ A LOB column cannot be used in the ORDER BY clause of a SELECT query.
- ⚠ A LOB column cannot appear in the GROUP BY clause of a SELECT query.
- ⚠ A LOB column cannot be used to join two tables.
- ⚠ The DISTINCT clause cannot be used with a LOB column in a SELECT query.
- ⚠ A LOB column cannot be selected in a UNION query.
- ⚠ A B-tree index or a bitmap index cannot be created on a LOB column.



Oracle LARGE Objects (LOBs)

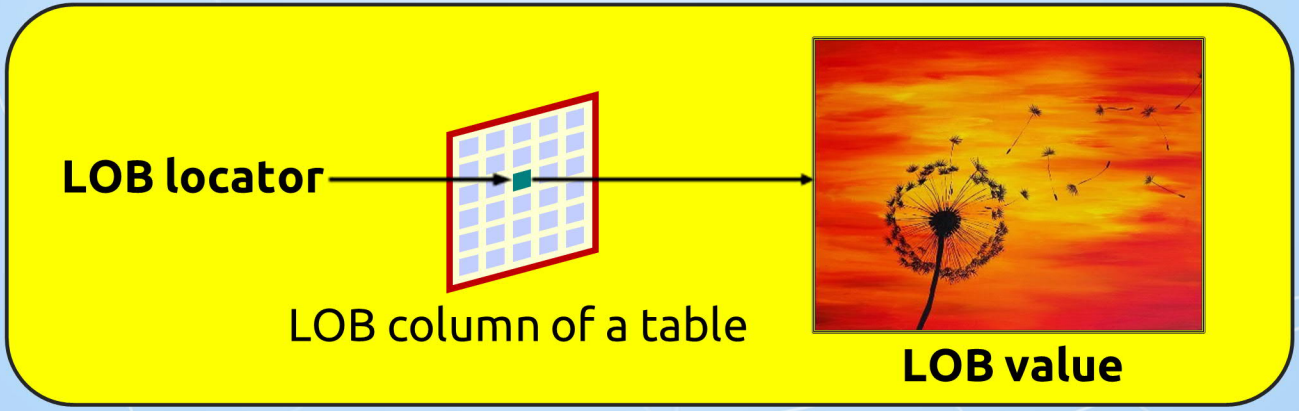
LOB Data Types



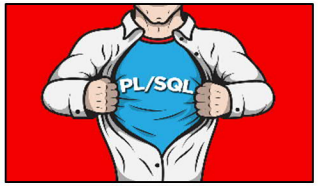
Oracle LARGE Objects (LOBs)

LOB Structure

- A LOB column stores an **LOB locator** and an **LOB value**.
- The LOB locator points to the location of the actual LOB file on the system.
- The LOB value is the actual data to be stored in the database.



- A LOB locator is always stored in a row irrespective of where the LOB value is stored.
- The LOB value is always stored in the **LOB segment**.



BLOB (Binary Large Object)

- BLOB is used to store binary large files such as images, audios, videos, PDFs, etc.
- BLOB stores 4 GB or more of unstructured or binary data. (Max: 128 TB)
- We can define attributes of an object type or columns of a database table as **BLOB**.
- We can have multiple columns that have BLOB data type{s} in the same table.

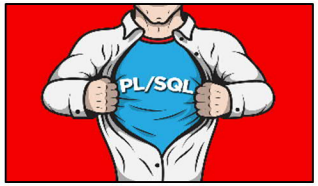
```
CREATE TABLE blob_table(
ID          NUMBER PRIMARY KEY,
photo       BLOB,
resume      BLOB);
```

- Assigning a value to a BLOB column;

1
```
INSERT INTO blob_table(ID, photo) VALUES(1, null);
```

2
```
INSERT INTO blob_table(ID, photo) VALUES(2, empty_blob());
```

3
```
INSERT INTO blob_table(ID, photo)
     VALUES (3, utl_raw.cast_to_raw('This is a sample text.'));
```



CLOB (Character Large Object)

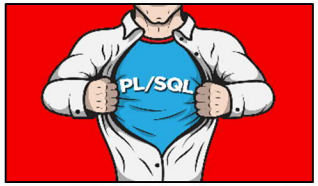
- CLOB is used to store 4GB of **character data** or more based on how your system is configured.
- It is used with the large set of characters, strings or documents that use the **database character set**.
- CLOBs are of two types. CLOB and NCLOB. They are similar.
- NCLOB(National Character Large Object) uses the National Character Set.
- You can define attributes of an object type or columns of a database as of type CLOB.
- You can have multiple columns of CLOB in the same table.

```
CREATE TABLE clob_table(
ID        NUMBER PRIMARY KEY,
resume    CLOB,
document  CLOB);
```

1
```
INSERT INTO clob_table(ID, document) VALUES(1, null);
```

2
```
INSERT INTO clob_table(ID, document) VALUES(2, empty_clob());
```

3
```
INSERT INTO clob_table(ID, document) VALUES(3,'This is a sample text.');
```



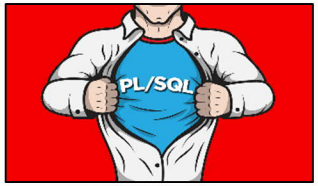
External Files (BFILEs)

- External LOBs are stored outside the database.
- Since the external LOB stores only the locations of the data, we need to work with a directory to associate with an OS file.
- We should first create a **directory** object.
- To create a directory;

1
```
CREATE DIRECTORY YOUR_DIRECTORY_NAME AS 'C:\full_path';
```

2
```
GRANT READ ON DIRECTORY YOUR_DIRECTORY_NAME TO hr;
```

- The files that you will use as BFILEs must reside on the machine where the database exists!
- If you migrate the DB to a different OS, you need to change the path value of the DIRECTORY.



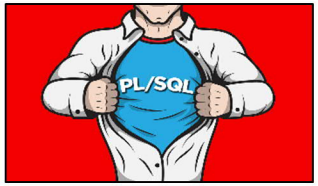
External Files (BFILEs)

- External LOBs are stored outside the database.
- Since the external LOB stores only the locations of the data, we need to work with a directory to associate with an OS file.
- We should first create a **directory** object.
- To create a directory;

1
```
CREATE DIRECTORY YOUR_DIRECTORY_NAME AS 'C:\full_path';
```

2
```
GRANT READ ON DIRECTORY YOUR_DIRECTORY_NAME TO hr;
```

- The files that you will use as BFILEs must reside on the machine where the database exists!
- If you migrate the DB to a different OS, you need to change the path value of the DIRECTORY.



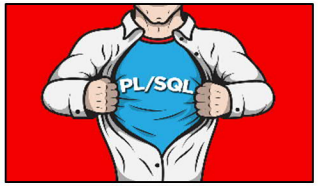
BFILE (Binary File)

CONTINUED...

- BFILES cannot participate in data operations or transactions.
- However, changes to a BFILE locator can be rolled back or committed.
- BFILEs can be used to load your external files in the operating system into your BLOB or CLOB columns.
- We use the BFILENAME built-in function to initialize a BFILE column

```
BFILENAME(directory_alias, file_name);
```

```
FUNCTION BFILENAME(directory_alias IN VARCHAR2,
                   filename IN VARCHAR2)
RETURN BFILE;
```

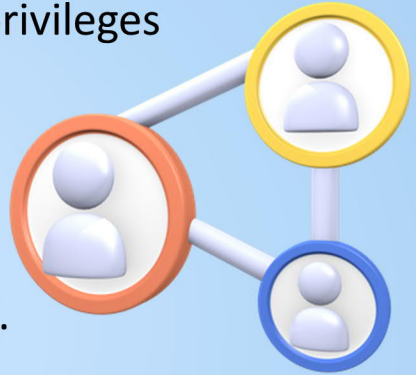


The DMBS_LOB() Package

- To work and fully interact with the LOBs, you may have to use very often the Oracle-supplied **DBMS_LOB** package.
- The DBMS_LOB package provides many functions and procedures to access and manipulate internal and external LOBs.

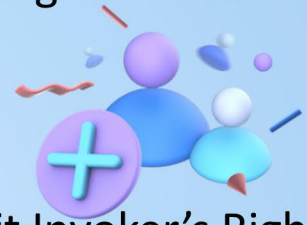
Definer's Rights

- Definer's Rights & Invoker's Rights are used to control access to the privileges necessary to run a user-defined program unit.
- Users don't need any other privilege than the **EXECUTE** privilege to run a Definer's Rights program unit.
- The **EXECUTE** privilege will not grant any other privileges to grantees.
- If any privilege necessary for a program unit is revoked from the program-unit owner, the program unit will be invalid.
- The **AUTHID DEFINER** keywords are used to make a subprogram Definer's Rights.
- By default, all program units are created as Definer's Rights.
- Definer's Rights give better control over the program units by preventing direct access to its underlying objects.



Invoker's Rights

- Users need to have the related privileges for all the objects inside the program units in addition to the **EXECUTE** privilege.
- Invoker's Rights are useful for increasing reuse of code.
- The **AUTHID CURRENT_USER** keywords are used to make a program unit Invoker's Rights.
- The **AUTHID DEFINER** and **AUTHID CURRENT_USER** keywords are used with packages, procedures, functions, views and types.
- Invoker's Rights program units should be created in a high-privileged schema.
- We can use **INHERIT PRIVILEGES** or **INHERIT ANY PRIVILEGES** to secure invoker's rights program units.
- All the users are granted **INHERIT PRIVILEGES** to **PUBLIC** by default.



Definer's & Invoker's Rights

```
REVOKE INHERIT [ANY] PRIVILEGES ON user FROM {subprogram_owner | PUBLIC};
```

```
GRANT INHERIT [ANY] PRIVILEGES TO {subprogram_owner | PUBLIC};
```

```
CREATE [OR REPLACE] VIEW view_name BEQUEATH {DEFINER|CURRENT_USER} AS ...;
```

☑ Code-Based Access Control (CBAC) can be used for a higher level of security on program units.

```
GRANT role[,...]] TO program_unit_type schema_name.program_unit_name[,...];
```