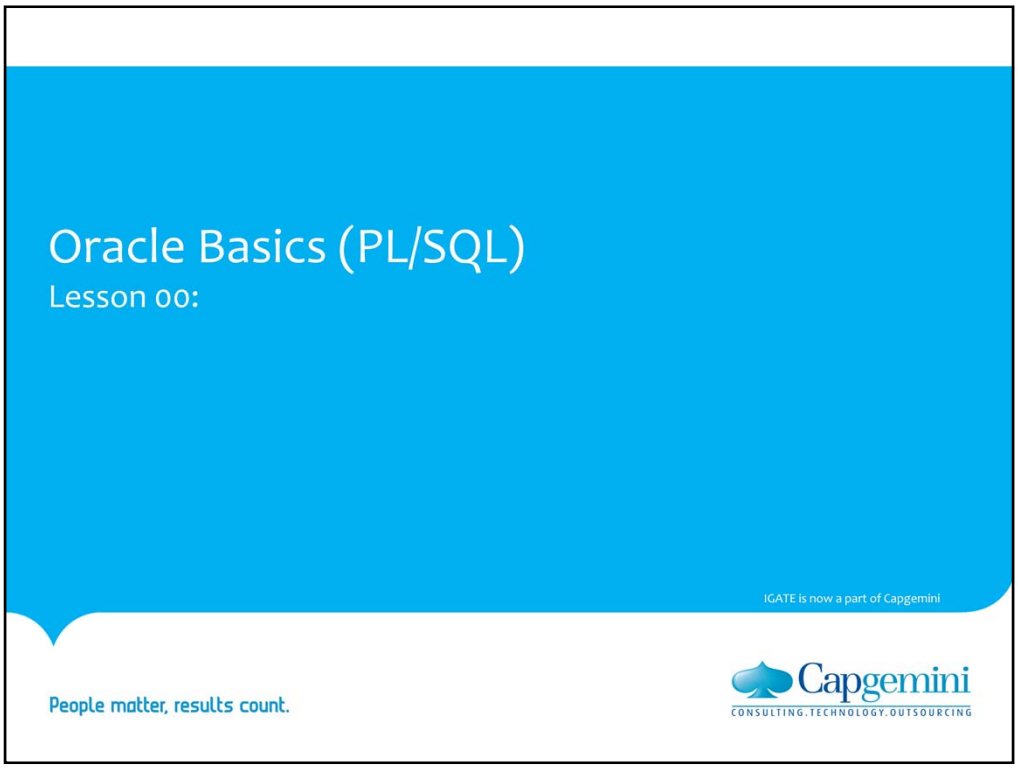


# Document History

| Date | Course Version No. | Software Version No. | Developer / SME | Change Record Remarks |
|---|---|---|---|---|
| 13-Nov-2008 | 1.0 | Oracle9i | Rajita Dhumal | Content Creation. |
| 28-Nov-2008 | 1.1 | Oracle9i | CLS team | Review. |
| 14-Jan-2010 | 1.2 | Oracle9i | Anu Mitra, | Review |
| 14-Jan-2010 | 1.2 | Oracle9i | Rajita Dhumal, CLS Team | Incorporating Review Comments |
| 16-May-2011 | 2.0 | Oracle 9i | Anu Mitra | Integration Refinements |
| 17-May-2013 | 2.1 | Oracle 9i | Hareshkumar Chandiramani | Courseware Refinements |
| 7-May-2016 | 2.2 | Oracle 9i | Kavita Arora | Integration Refinements |

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 2

## Course Goals and Non Goals

- Course Goals
  - To understand RDBMS Methodology.
  - To code PL/SQL Blocks for Implementing business rules.
  - To create Stored Subprograms using Procedures and Functions.
  - To implement Business Rule using Constraints.
- Course Non Goals
  - Object Oriented programming concepts (ORDBMS) are not covered as a part of this course.

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 3

## Pre-requisites

- Require a fair proficiency level in Relational Database Concepts.
- Require good proficiency in DBMS SQL

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

## Intended Audience

- Software Programmers
- Software Analysts

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 5

## Day Wise Schedule

- Day 1
  - Lesson 1: PL/SQL Basics
  - Lesson 2: Introduction to Cursors
- Day 2
  - Lesson 3: Exception Handling
  - Lesson 4: Procedures and Functions

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 6

# Table of Contents



Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

# Table of Contents



Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 8

# Table of Contents



Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 9

## References

- Oracle PL/SQL Programming, Third Edition; by Steven Feuerstein
- Oracle 9i PL/SQL: A Developer's Guide
- Oracle9i PL/SQL Programming; by Scott Urman

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 10

## Next Step Courses (if applicable)

- Data Warehousing Concepts
- Reporting / ETL tools
- Database Administration / Database Performance Tuning

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 11

## Other Parallel Technology Areas

- Microsoft SQL Server
- IBM DB2

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 12

# Oracle Basics (PL/SQL)

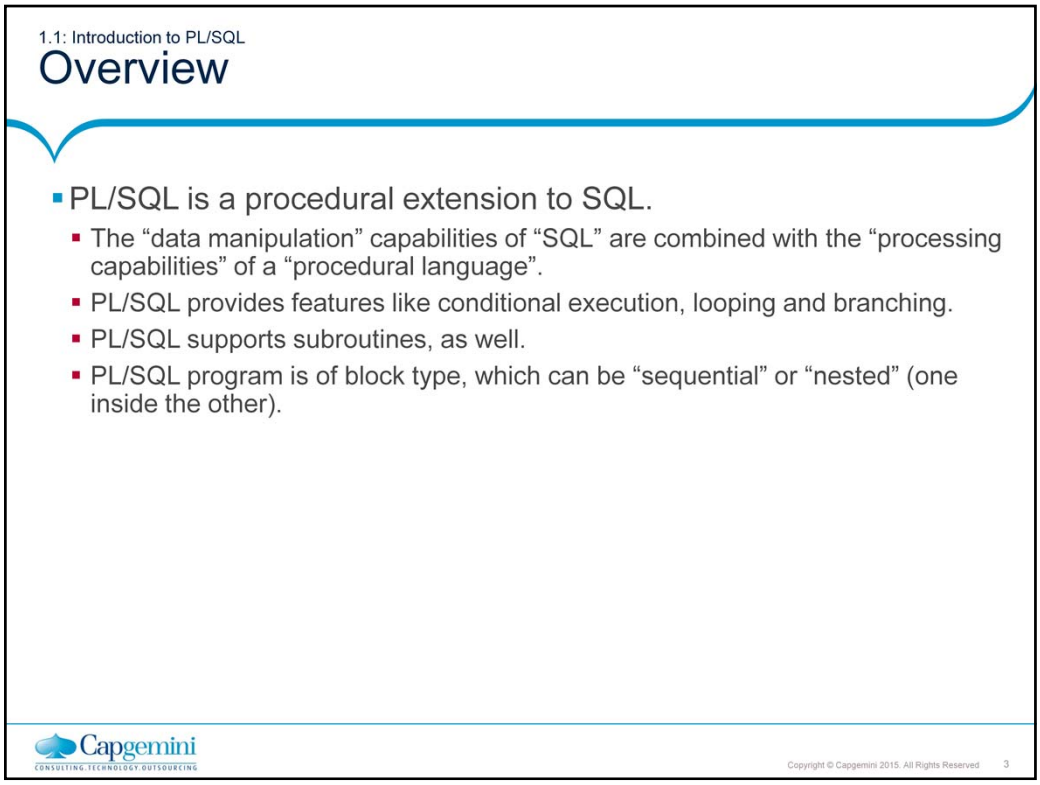
Lesson 01 Introduction to PL/SQL

## Lesson Objectives

- To understand the following topics:
  - Introduction to PL/SQL
  - PL/SQL Block structure
  - Handling variables in PL/SQL
  - Variable scope and Visibility
  - SQL in PL/SQL
  - Programmatic Constructs

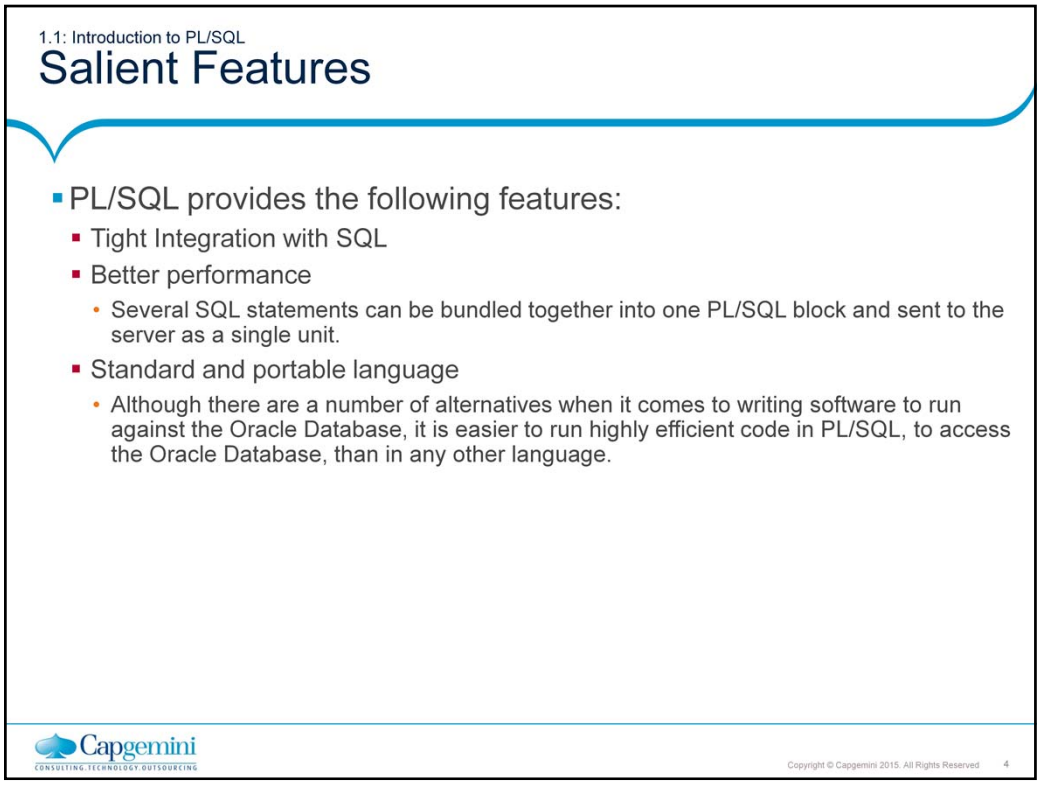
Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 2



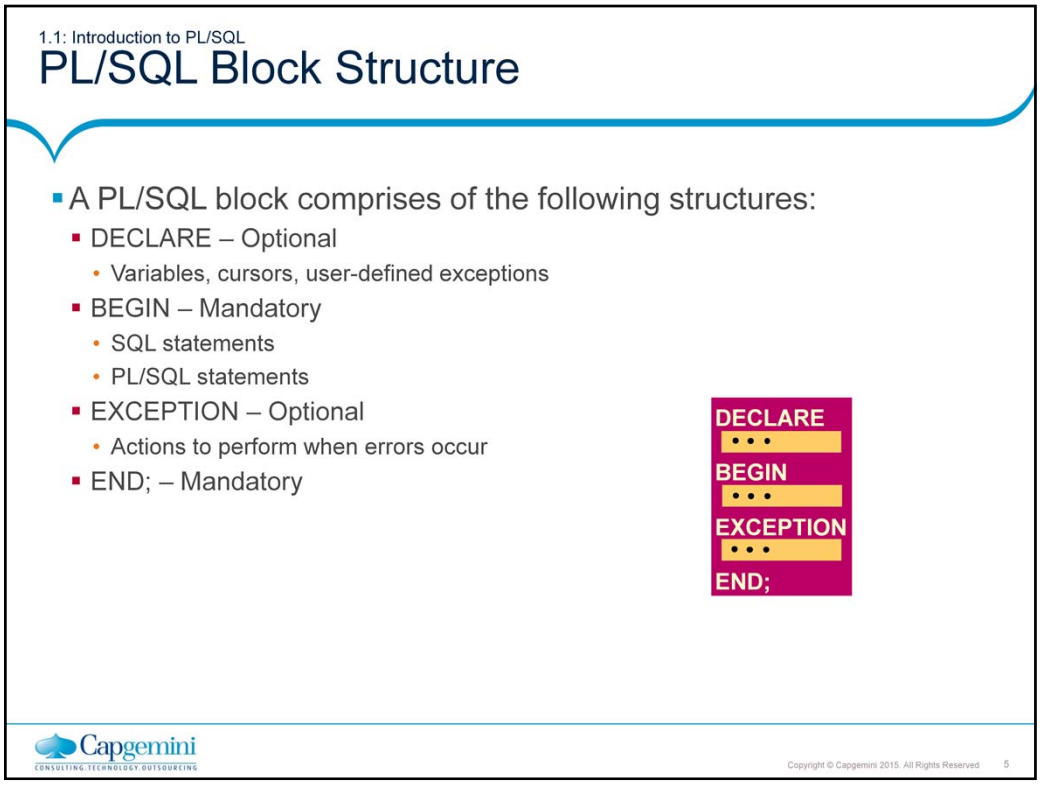

**Introduction to PL/SQL:**

- PL/SQL stands for Procedural Language/SQL. PL/SQL extends SQL by adding constructs found in procedural languages, resulting in a structural language that is “more powerful than SQL”.
  - With PL/SQL, you can use SQL statements to manipulate Oracle data and flow-of-control statements to process the data.
  - Moreover, you can declare constants and variables, define procedures and functions, and trap runtime errors.
  - Thus PL/SQL combines the “data manipulating power” of SQL with the “data processing power” of procedural languages.
- PL/SQL is an “embedded language”. It was not designed to be used as a “standalone” language but instead to be invoked from within a “host” environment.
  - You cannot create a PL/SQL “executable” that runs all by itself.
  - It can run from within the database through SQL*Plus interface or from within an Oracle Developer Form (called client-side PL/SQL).




**Features of PL/SQL**

- Tight Integration with SQL:
  - This integration saves both, your learning time as well as your processing time.
    - PL/SQL supports SQL data types, reducing the need to convert data passed between your application and database.
    - PL/SQL lets you use all the SQL data manipulation, cursor control, transaction control commands, as well as SQL functions, operators, and pseudo columns.
- Better Performance:
  - Several SQL statements can be bundled together into one PL/SQL block, and sent to the server as a single unit.
  - This results in less network traffic and a faster application. Even when the client and the server are both running on the same machine, the performance is increased. This is because packaging SQL statements results in a simpler program that makes fewer calls to the database.
- Portable:
  - PL/SQL is a standard and portable language.
  - A PL/SQL function or procedure written from within the Personal Oracle database on your laptop will run without any modification on your corporate network database. It is “Write once, run everywhere” with the only restriction being “everywhere” there is an Oracle Database.
- Efficient:
  - Although there are a number of alternatives when it comes to writing software to run against the Oracle Database, it is easier to run highly efficient code in PL/SQL, to access the Oracle Database, than in any other language.



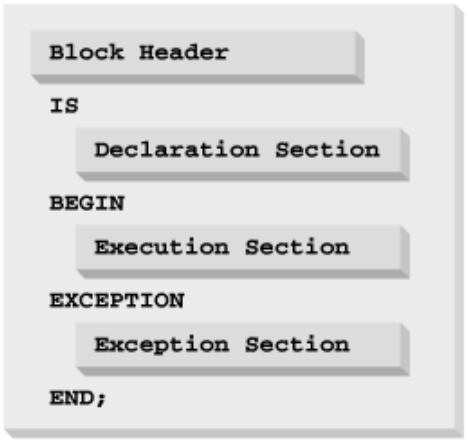

**PL/SQL Block Structure:**

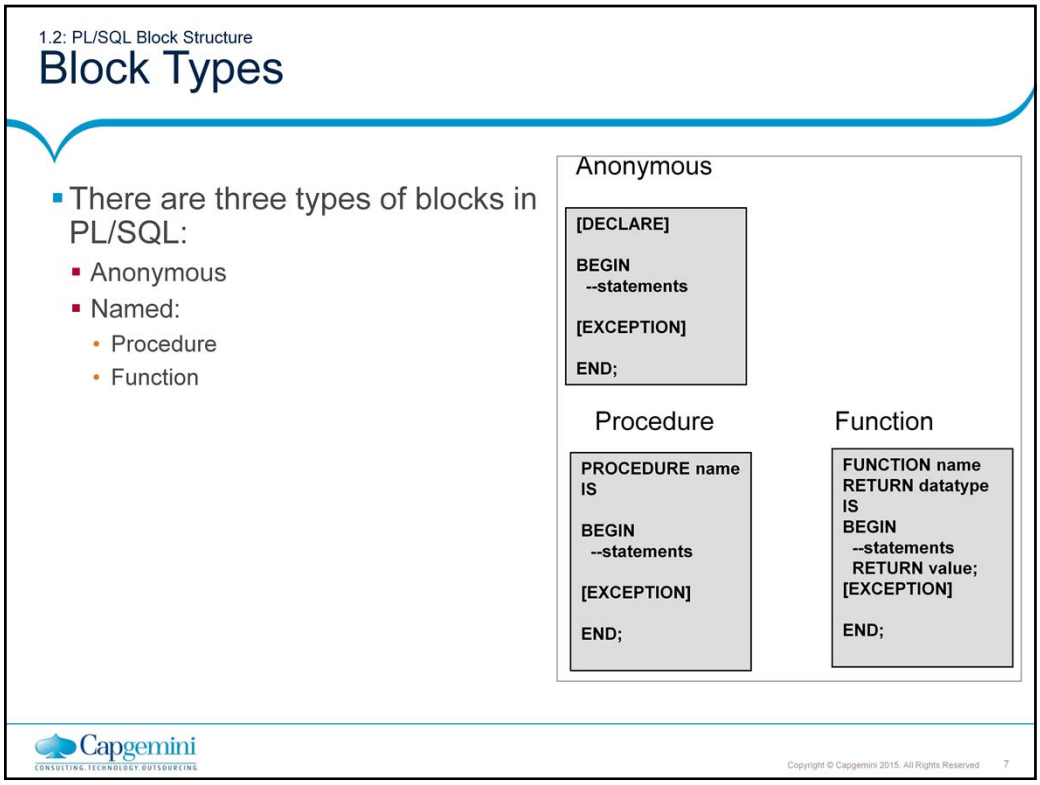
- PL/SQL is a block-structured language. Each basic programming unit that is written to build your application is (or should be) a “logical unit of work”. The PL/SQL block allows you to reflect that logical structure in the physical design of your programs.
- Each PL/SQL block has up to four different sections (some are optional under certain circumstances).

contd.

**PL/SQL block structure (contd.):**

- **Header**
  It is relevant for named blocks only. The header determines the way the named block or program must be called.
- **Declaration section**
  The part of the block that declares variables, cursors, and sub-blocks that are referenced in the Execution and Exception sections.
- **Execution section**
  It is the part of the PL/SQL blocks containing the executable statements; the code that is executed by the PL/SQL runtime engine.
- **Exception section**
  It is the section that handles exceptions for normal processing (warnings and error conditions).





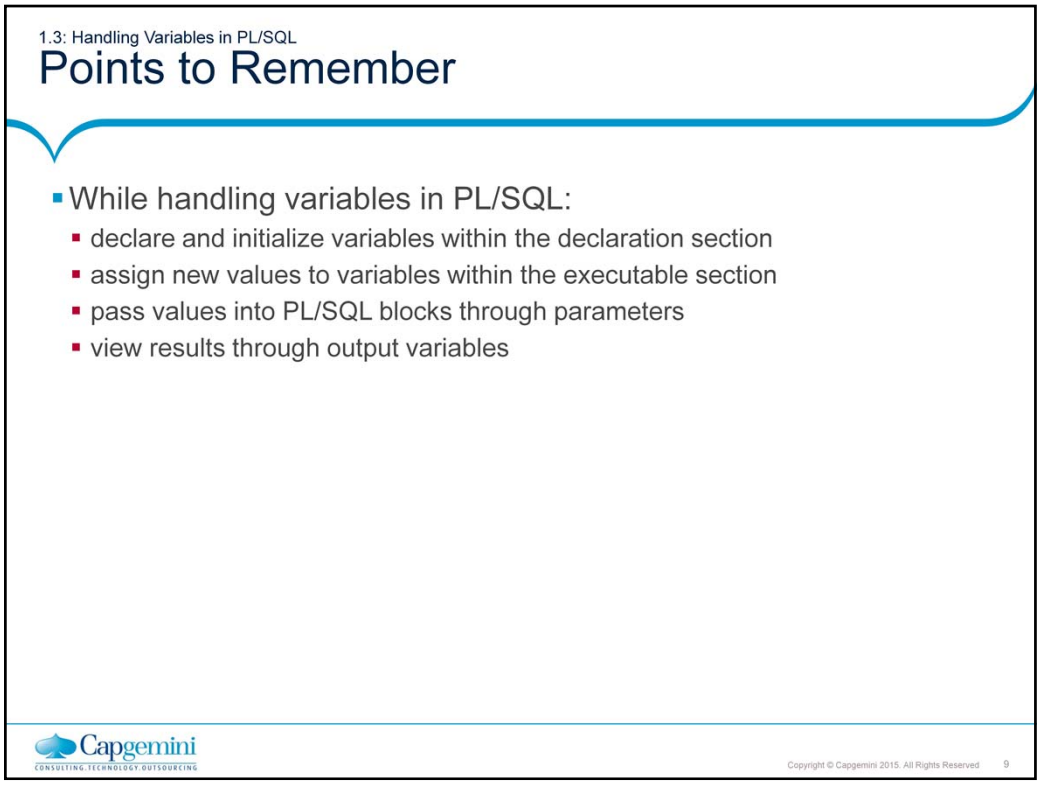

**Block Types:**

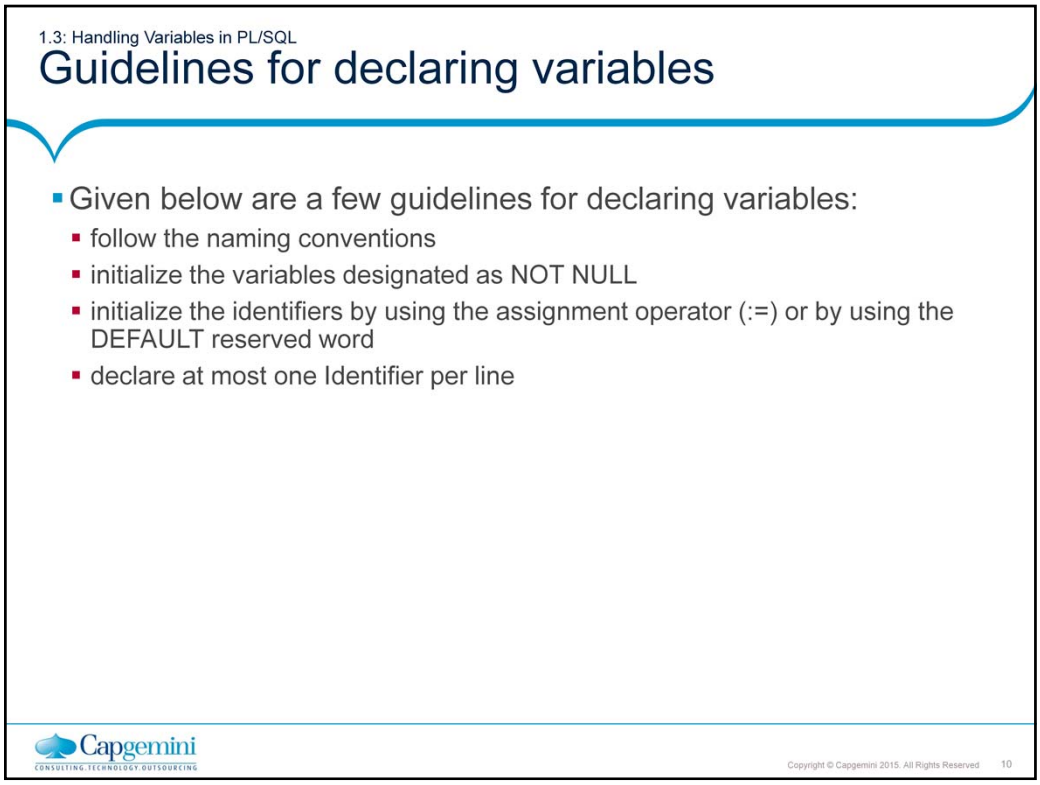
- The basic units (procedures and functions, also known as subprograms, and anonymous blocks) that make up a PL/SQL program are “logical blocks”, which can contain any number of nested sub-blocks.
- Therefore one block can represent a small part of another block, which in turn can be part of the whole unit of code.
  - **Anonymous Blocks**
    Anonymous blocks are unnamed blocks. They are declared at the point in an application where they are to be executed and are passed to the PL/SQL engine for execution at runtime.
  - **Named :**
    - **Subprograms**
      Subprograms are named PL/SQL blocks that can take parameters and can be invoked. You can declare them either as “procedures” or as “functions”.
      Generally, you use a “procedure” to perform an “action” and a “function” to compute a “value”.

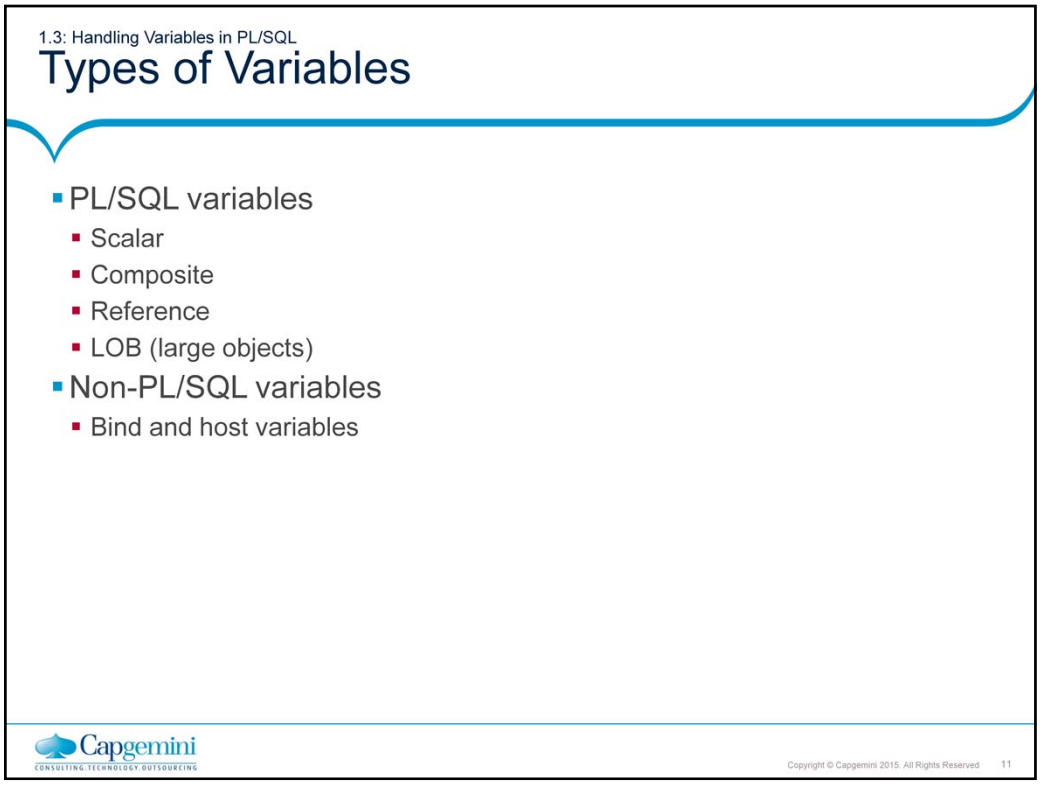
**Representation of a PL/SQL block:**

- The notations used in a PL/SQL block are given below:
  1. -- is a single line comment.
  2. /* */ is a multi-line comment.
  3. Every statement must be terminated by a semicolon (;).
  4. PL/SQL block is terminated by a slash (/) on a line by itself.
- A PL/SQL block must have an “Execution section”.
- It can optionally have a “Declaration section” and an Exception section, as well.

```
DECLARE                    -- Declaration Section
  V_Salary    NUMBER(7,2);
/* V_Salary is a variable declared in a PL/SQL block. This variable is
used to store JONES' salary. */
Low_Sal  EXCEPTION;                    -- an exception
BEGIN                          -- Execution Section
SELECT sal INTO V_Salary
FROM emp WHERE ename = 'JONES'
        IF V_Salary < 3000 THEN
                RAISE Low_Sal ;
        END IF;
EXCEPTION                      -- Exception Section
    WHEN Low_Sal     THEN
    UPDATE emp SET sal = sal + 500 WHERE ename = 'JONES' ;
END ;                          -- End of Block
/                              -- PL/SQL block terminator
Output
SQL> /
PL/SQL procedure successfully completed.
```

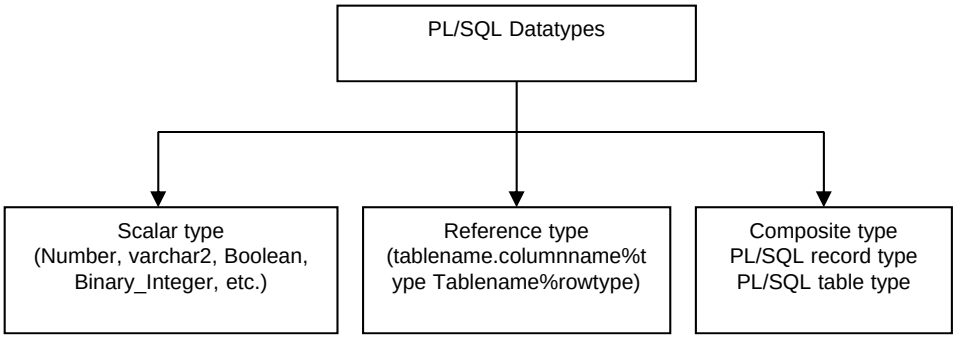

1.3: Handling Variables in PL/SQL
Points to Remember
While handling variables in PL/SQL:
declare and initialize variables within the declaration section
assign new values to variables within the executable section
pass values into PL/SQL blocks through parameters
view results through output variables
Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING
Copyright © Capgemini 2015. All Rights Reserved
9

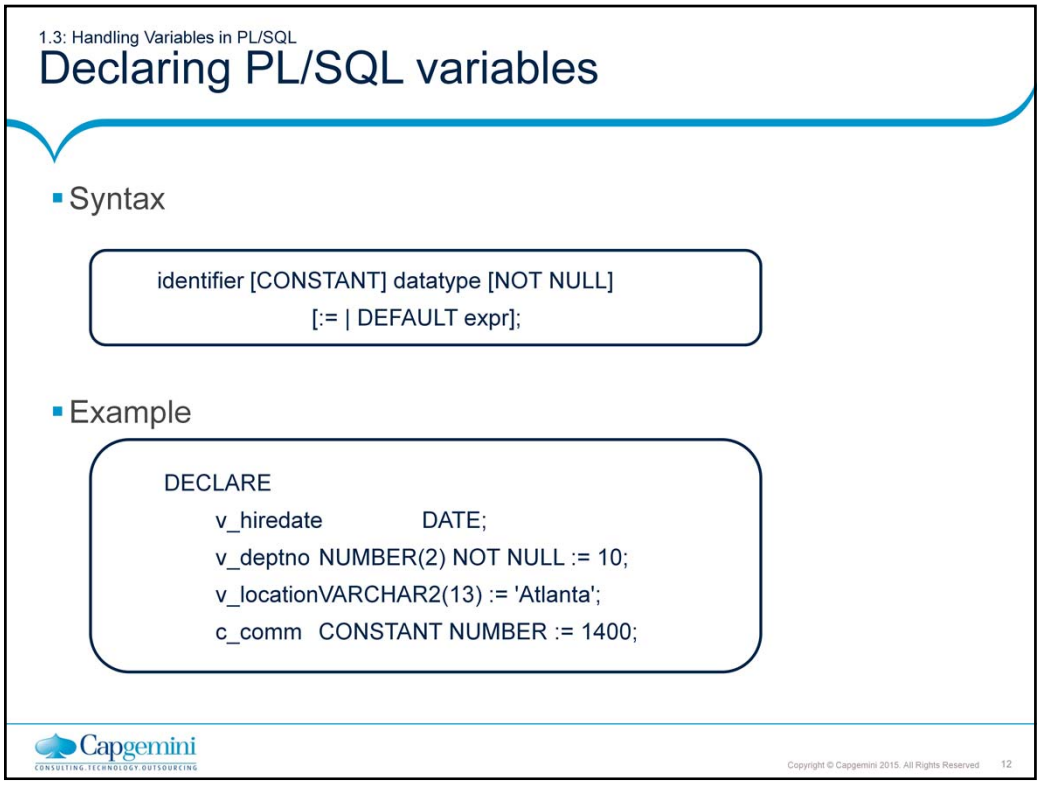

1.3: Handling Variables in PL/SQL
Guidelines for declaring variables
Given below are a few guidelines for declaring variables:
follow the naming conventions
initialize the variables designated as NOT NULL
initialize the identifiers by using the assignment operator (:=) or by using the DEFAULT reserved word
declare at most one Identifier per line
Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING
Copyright © Capgemini 2015. All Rights Reserved
10




**Types of Variables: PL/SQL Datatype:**

- All PL/SQL datatypes are classified as scalar, reference and Composite type.
- Scalar datatypes do not have any components within it, while composite datatypes have other datatypes within them.
- A reference datatype is a pointer to another datatype.






**Declaring PL/SQL Variables:**

- You need to declare all PL/SQL identifiers within the "declaration section" before referencing them within the PL/SQL block.
- You have the option to assign an initial value.
  - You do not need to assign a value to a variable in order to declare it.
  - If you refer to other variables in a declaration, you must separately declare them in a previous statement.
  - Syntax:

```
identifier [CONSTANT] datatype [NOT NULL]
                    [:= | DEFAULT expr];
```

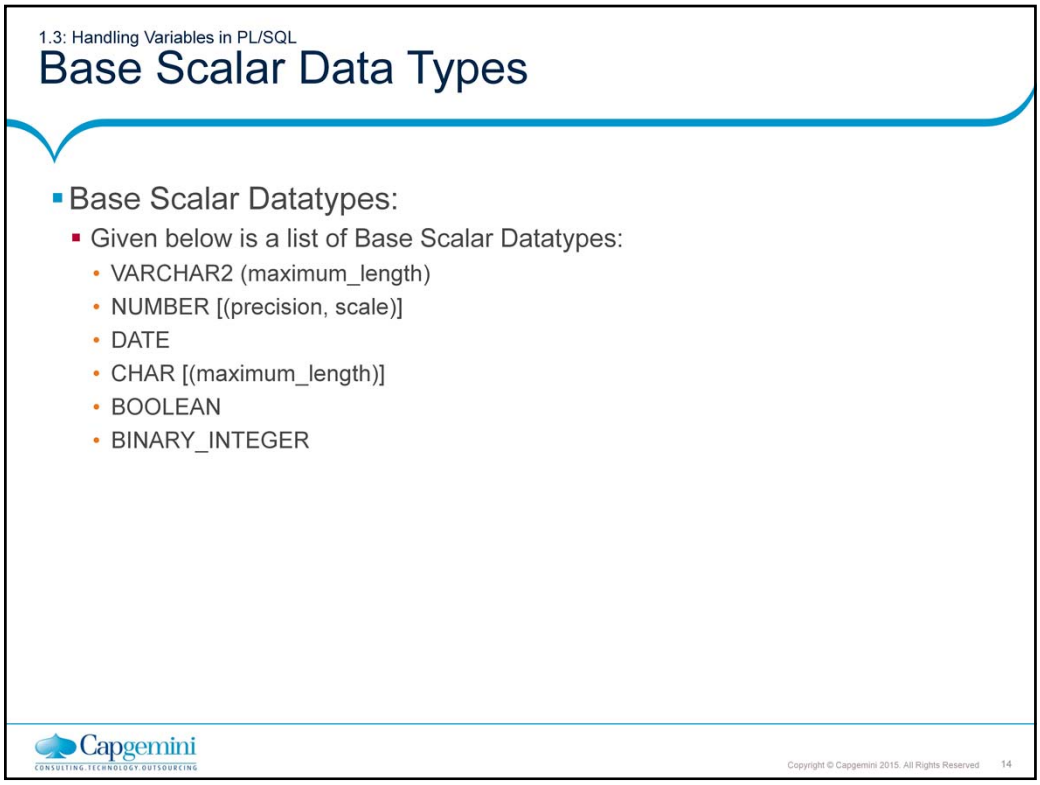
  - In the syntax given above:
    - **identifier** is the name of the variable.
    - **CONSTANT** constrains the variable so that its value cannot change. Constants must be initialized.
    - **datatype** is a scalar, composite, reference, or LOB datatype.
    - **NOT NULL** constrains the variable so that it must contain a value. NOT NULL variables must be initialized.
    - **expr** is any PL/SQL expression that can be a literal, another variable, or an expression involving operators and functions.

contd.

**Declaring PL/SQL Variables (contd.):**

**For example:**

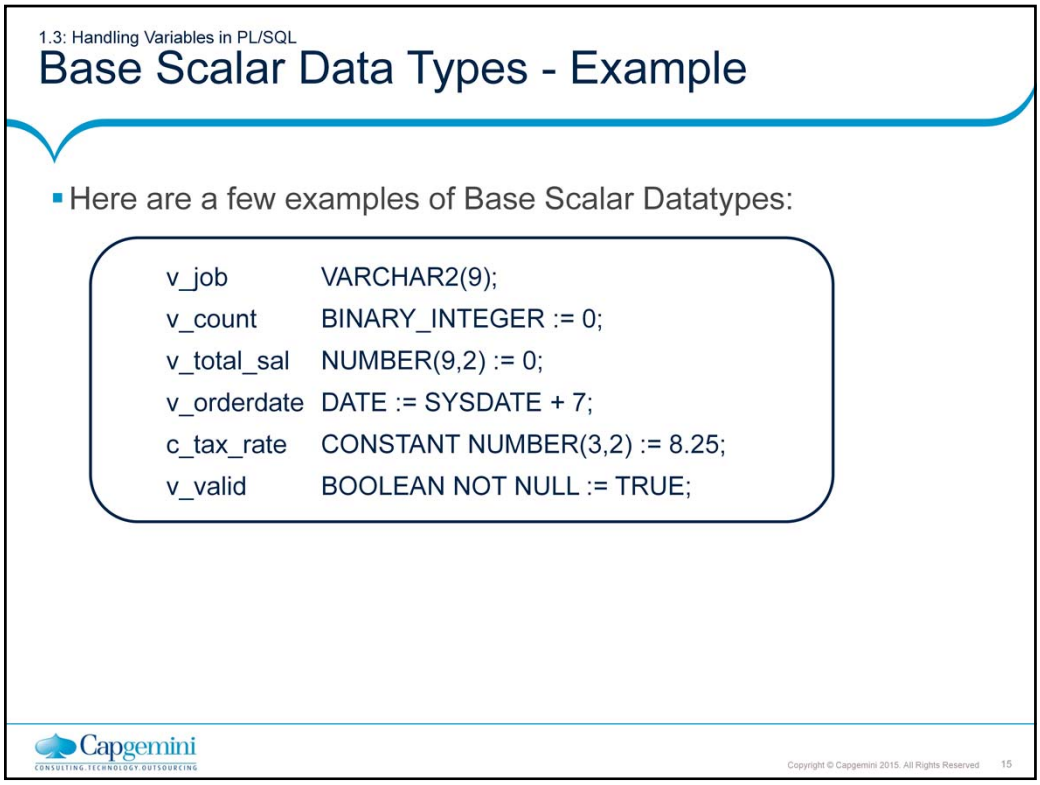
```
DECLARE
   v_description varchar2 (25);
   v_sal             number (5) not null: = 3000;
   v_compcode  varchar2 (20) constant: = 'abc
                        consultants';
   v_comm        not null default   0;
```




**Base Scalar Datatypes:**

1. **NUMBER**
   This can hold a numeric value, either integer or floating point. It is same as the number database type.
2. **BINARY_INTEGER**
   If a numeric value is not to be stored in the database, the BINARY_INTEGER datatype can be used. It can only store integers from -2147483647 to + 2147483647. It is mostly used for counter variables.
   V_Counter BINARY_INTEGER DEFAULT 0;
3. **VARCHAR2 (L)**
   L is necessary and is max length of the variable. This behaves like VARCHAR2 database type. The maximum length in PL/SQL is 32,767 bytes whereas VARCHAR2 database type can hold max 2000 bytes. If a VARCHAR2 PL/SQL column is more than 2000 bytes, it can only be inserted into a database column of type LONG.
4. **CHAR (L)**
   Here L is the maximum length. Specifying length is optional. If not specified, the length defaults to 1. The maximum length of CHAR PL/SQL variable is 32,767 bytes, whereas the maximum length of the database CHAR column is 255 bytes. Therefore a CHAR variable of more than 255 bytes can be inserted in the database column of VARCHAR2 or LONG type.

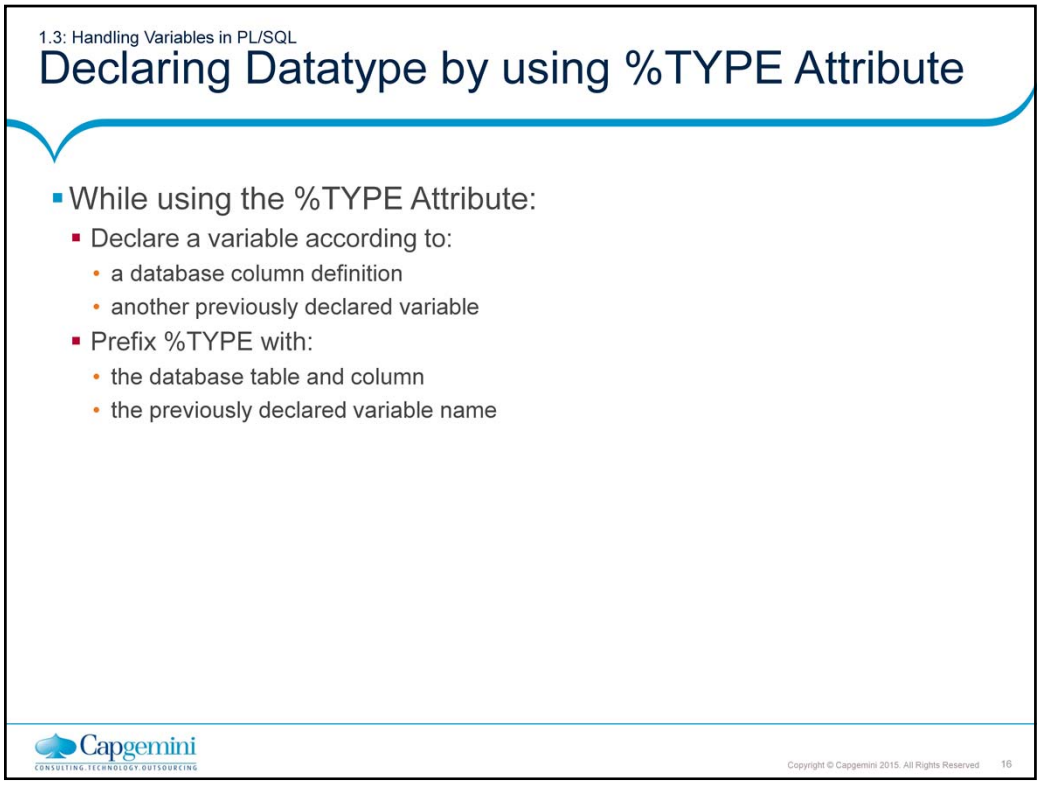
contd.




**Base Scalar Datatypes (contd.):**

5. **LONG**
   PL/SQL LONG type is just 32,767 bytes. It behaves similar to LONG DATABASE type.
6. **DATE**
   The DATE PL/SQL type behaves the same way as the date database type. The DATE type is used to store both date and time. A DATE variable is 7 bytes in PL/SQL.
7. **BOOLEAN**
   A Boolean type variable can only have one of the two values, i.e. either TRUE or FALSE. They are mostly used in control structures.
   V_Does_Dept_Exist BOOLEAN := TRUE;
   V_ Flag BOOLEAN := 0; -- illegal

```
declare
        pie constant number := 7.18;
        radius number := &radius;
begin
        dbms_output.put_line('Area:
'||pie*power(radius,2));
        dbms_output.put_line('Diameter: '||2*pie*radius);
end;
/
```




**Reference types:**

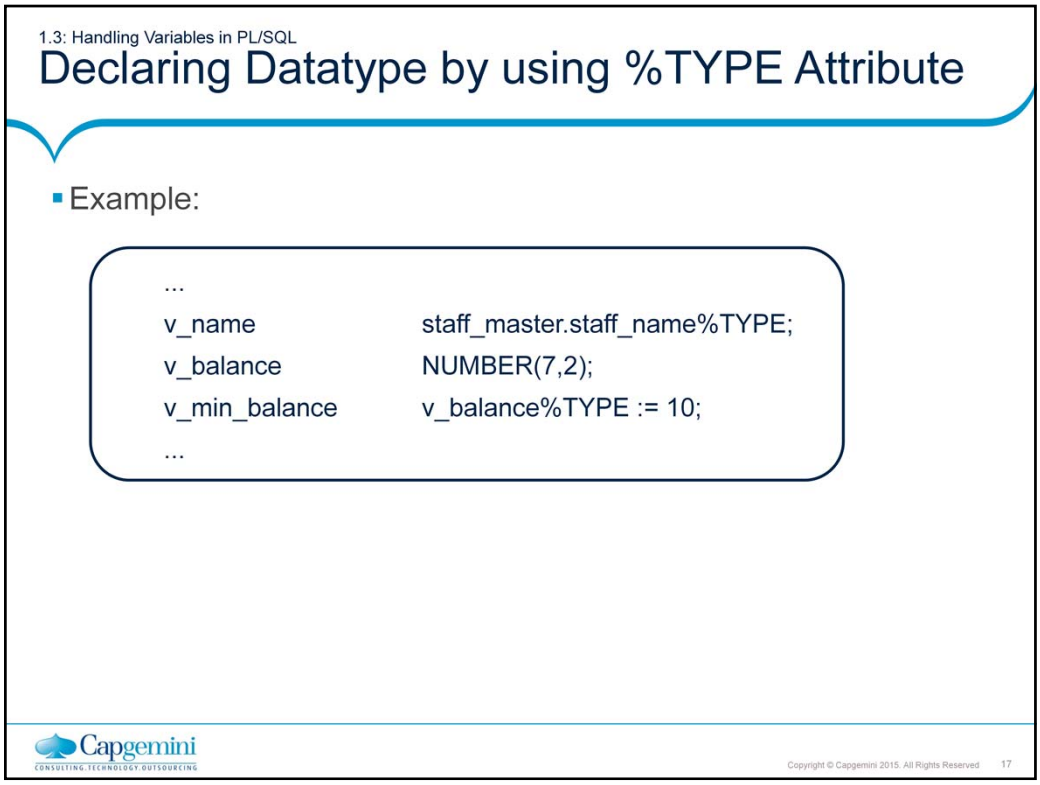
- A "reference type" in PL/SQL is the same as a "pointer" in C. A "reference type" variable can point to different storage locations over the life of the program.

**Using %TYPE**

- %TYPE is used to declare a variable with the same datatype as a column of a specific table. This datatype is particularly used when declaring variables that will hold database values.
- **Advantage:**
  - You need not know the exact datatype of a column in the table in the database.
  - If you change database definition of a column, it changes accordingly in the PL/SQL block at run time.
  - Syntax:

```
Var_Name        table_name.col_name%TYPE;
V_Empno         emp.empno%TYPE;
```

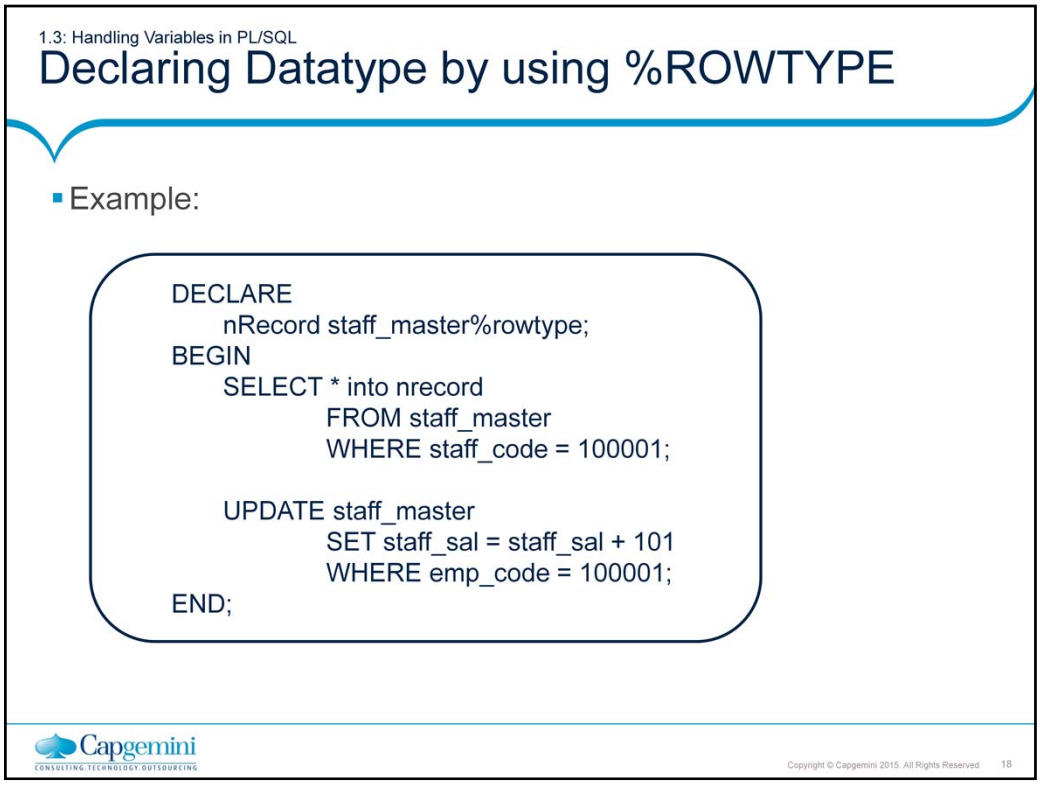
  - **Note:** Datatype of V_Empno is same as datatype of Empno column of the EMP table.




**Using %TYPE (contd.)**

- Example

```
declare
        nSalary employee.salary%type;
begin
      select salary into nsalary
        from employee
        where emp_code = 11;
        update employee set salary = salary + 101
where emp_code = 11;
end;
```



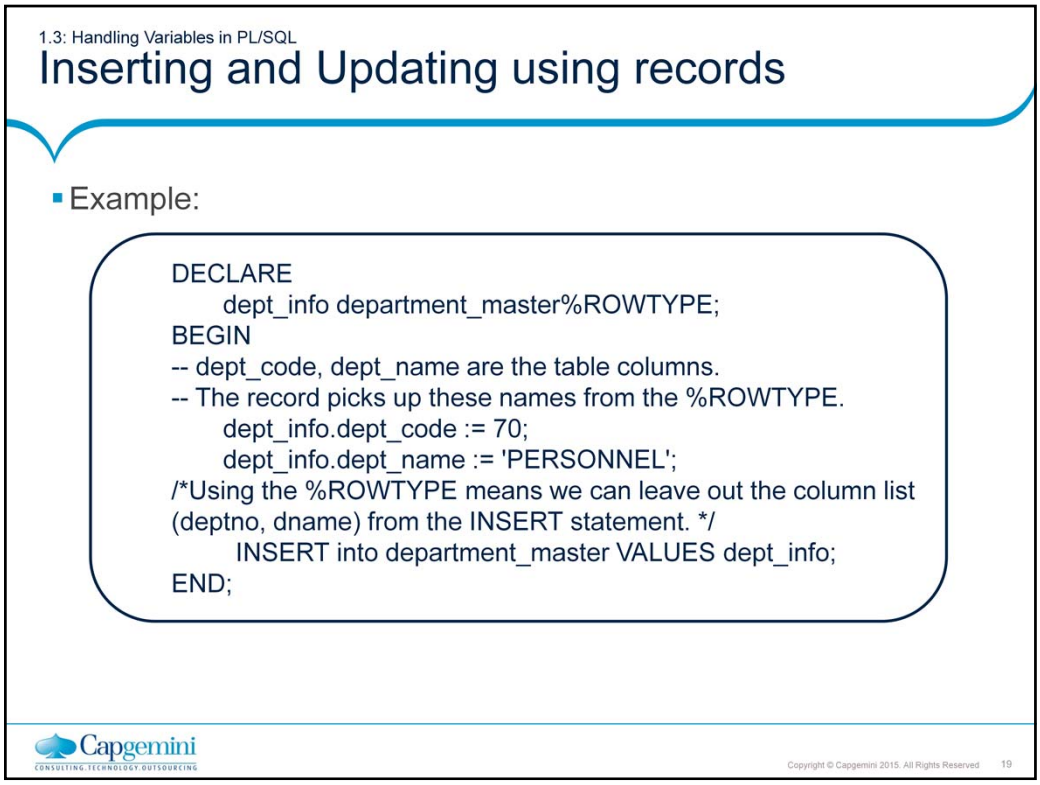

**Using %ROWTYPE**

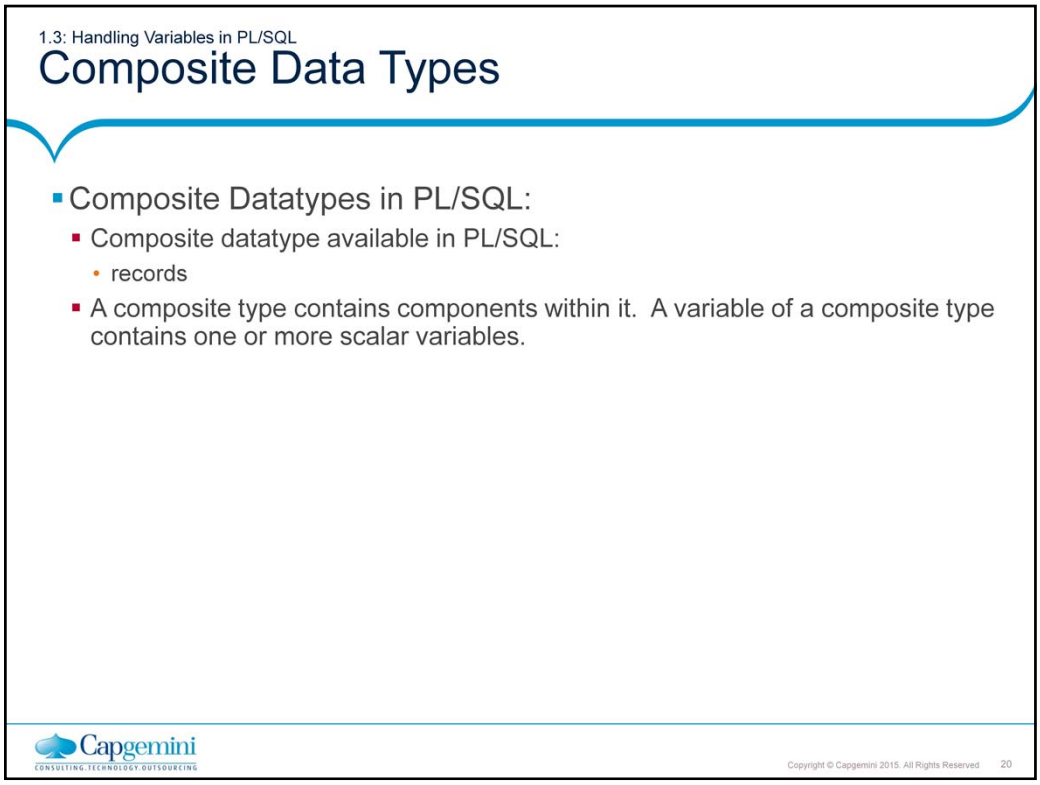
- %ROWTYPE is used to declare a compound variable, whose type is same as that of a row of a table.
- Columns in a row and corresponding fields in record should have same names and same datatypes. However, fields in a %ROWTYPE record do not inherit constraints, such as the NOT NULL, CHECK constraints, or default values.
- **Syntax:**
- V_Emprec emp%rowtype

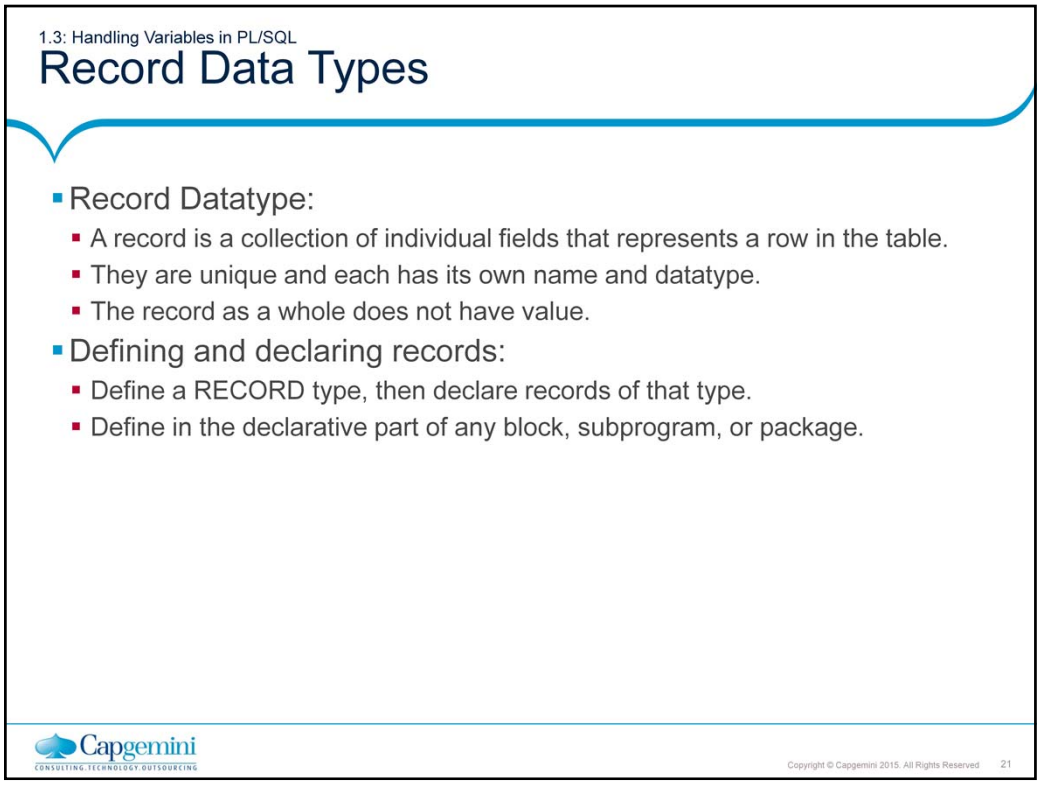
```
Var_Name        table_name%ROWTYPE;
V_Emprec        emp%ROWTYPE;
```

- where V_Emprec is a variable, which contains within itself as many variables, whose names and datatypes match those of the EMP table.
  - To access the Empno element of V_Emprec, use V_Emprec.empno;

```
DECLARE emprec emp%rowtype;
BEGIN
emprec.empno :=null;
emprec.deptno :=50;
dbms_output.put_line ('emprec.employee's
number'||emprec.empno);
END;
/
```

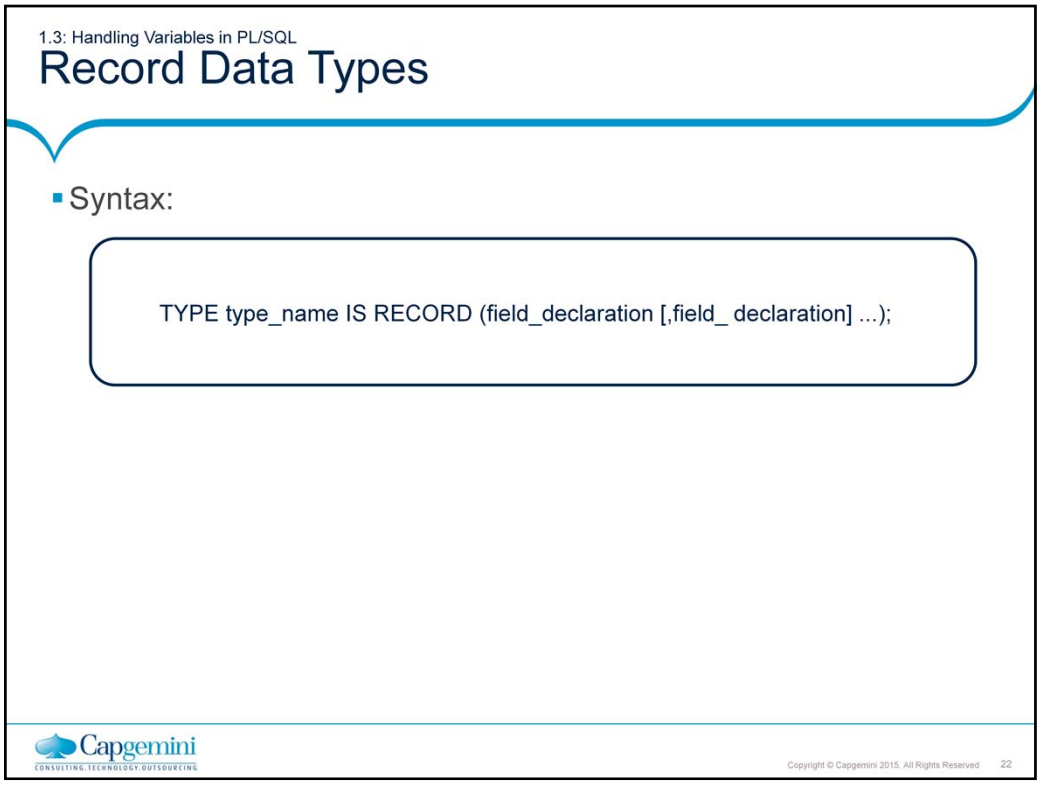

1.3: Handling Variables in PL/SQL
Inserting and Updating using records
Example:
DECLARE
dept_info department_master%ROWTYPE;
BEGIN
-- dept_code, dept_name are the table columns.
-- The record picks up these names from the %ROWTYPE.
dept_info.dept_code := 70;
dept_info.dept_name := 'PERSONNEL';
/*Using the %ROWTYPE means we can leave out the column list
(deptno, dname) from the INSERT statement. */
INSERT into department_master VALUES dept_info;
END;
Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING
Copyright © Capgemini 2015. All Rights Reserved
19


1.3: Handling Variables in PL/SQL
Composite Data Types
Composite Datatypes in PL/SQL:
Composite datatype available in PL/SQL:
records
A composite type contains components within it. A variable of a composite type contains one or more scalar variables.
Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING
Copyright © Capgemini 2015. All Rights Reserved 20



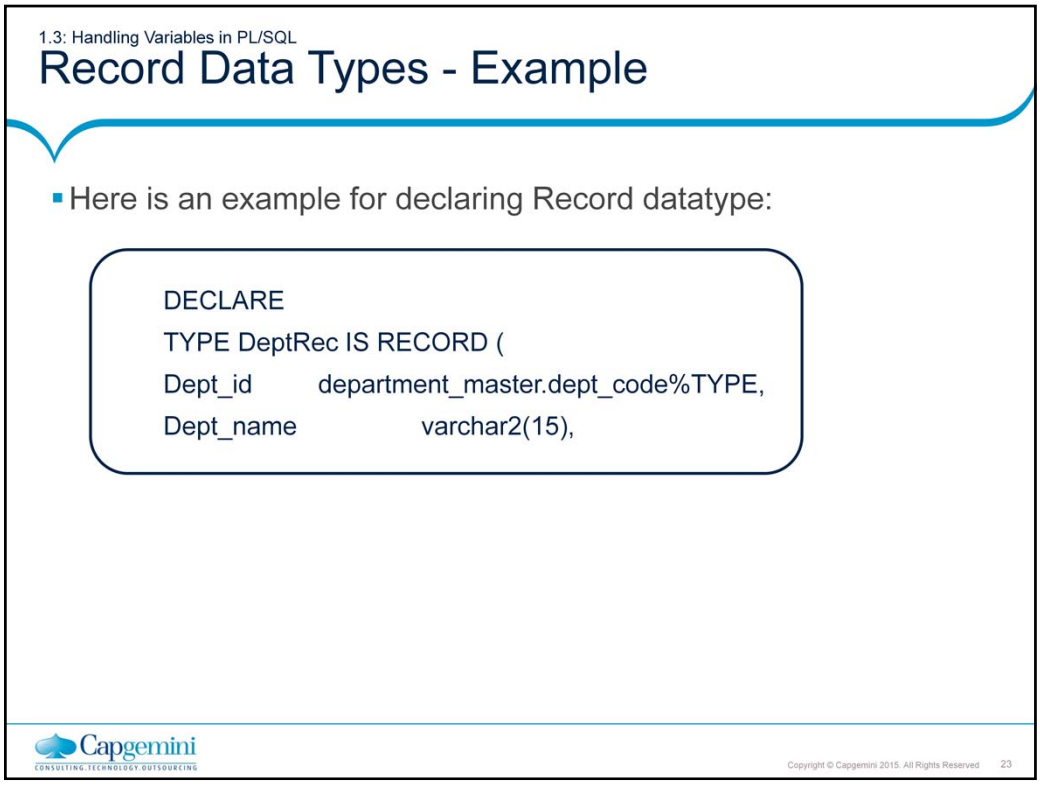

**Record Datatype:**

- A record is a collection of individual fields that represents a row in the table. They are unique and each has its own name and datatype. The record as a whole does not have value. By using records you can group the data into one structure and then manipulate this structure into one “entity” or “logical unit”. This helps to reduce coding and keeps the code easier to maintain and understand.



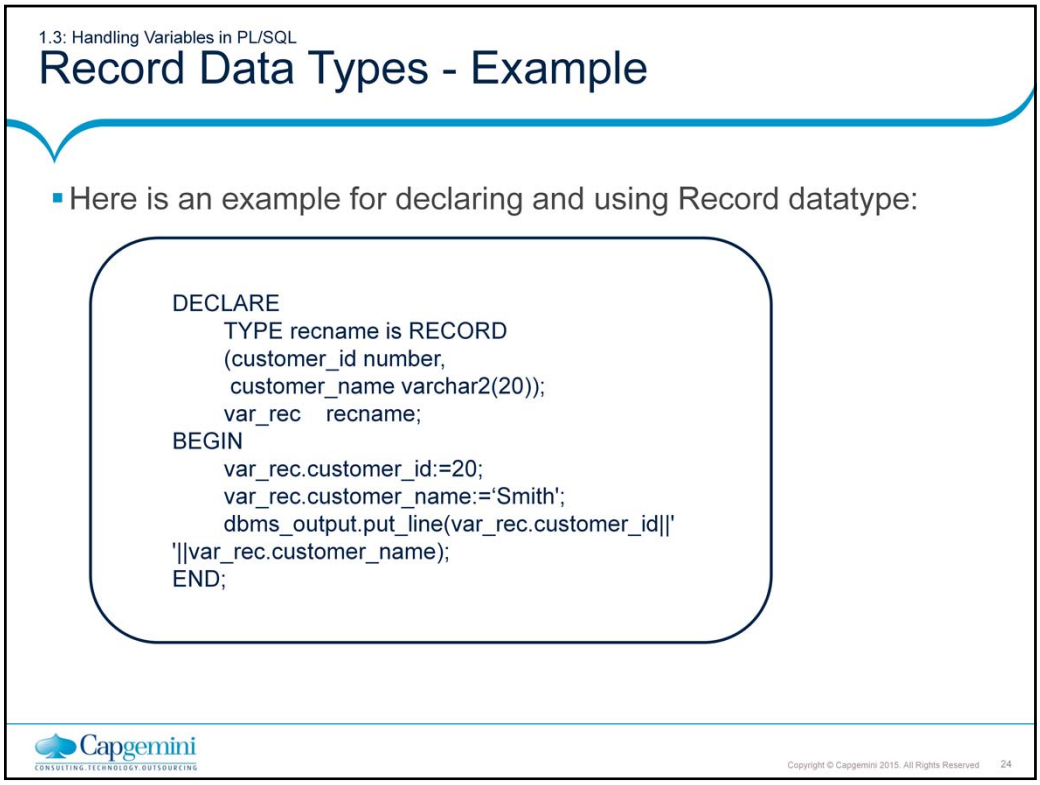

**Defining and Declaring Records**

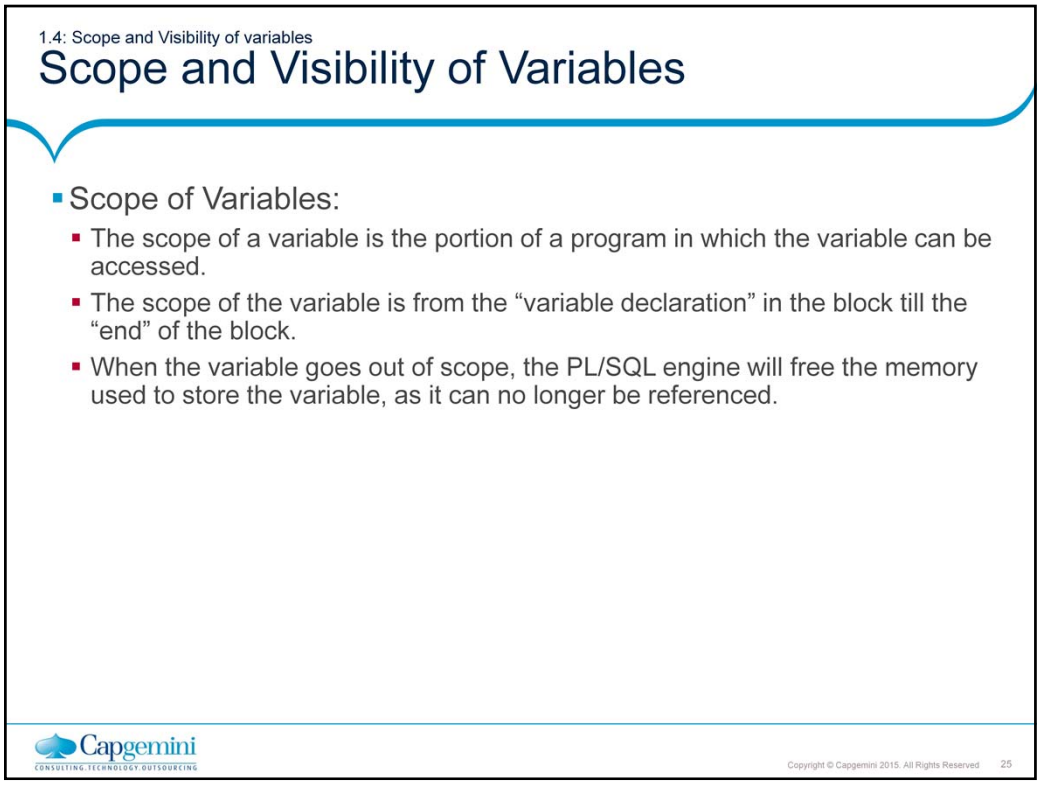
- To create records, you define a RECORD type, then declare records of that type. You can define RECORD types in the declarative part of any PL/SQL block, subprogram, or package by using the syntax.
- where field_declaration stands for:
  - •field_name field_type [[NOT NULL] {:= | DEFAULT} expression]
  - •type_name is a type specifier used later to declare records. You can use %TYPE and %ROWTYPE to specify field types.




**Record Datatype (contd.):**

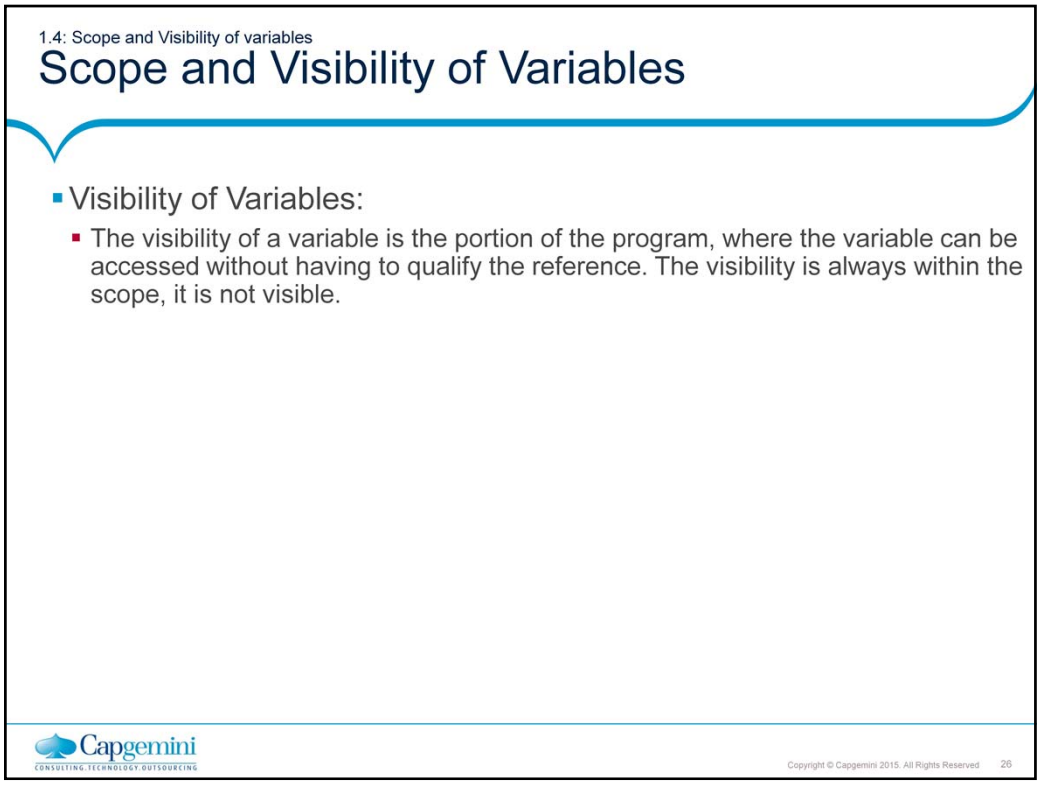
- **Field declarations** are like variable declarations.
- Each field has a unique name and specific datatype.
- Record members can be accessed by using "." (Dot) notation.
- The value of a record is actually a collection of values, each of which is of some simpler type. The attribute %ROWTYPE lets you declare a record that represents a row in a database table.
- After a record is declared, you can reference the record members directly by using the "." (Dot) notation. You can reference the fields in a record by indicating both the record and field names.
  **For example**: To reference an individual field, you use the dot notation DeptRec.deptno;
- You can assign expressions to a record.
  **For example**: DeptRec.deptno := 50;
- You can also pass a record type variable to a procedure as shown below: get_dept(DeptRec);


1.3: Handling Variables in PL/SQL
Record Data Types - Example
Here is an example for declaring and using Record datatype:
DECLARE
TYPE recname is RECORD
(customer_id number,
customer_name varchar2(20));
var_rec recname;
BEGIN
var_rec.customer_id:=20;
var_rec.customer_name:='Smith';
dbms_output.put_line(var_rec.customer_id||'
'||var_rec.customer_name);
END;
Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING
Copyright © Capgemini 2015. All Rights Reserved
24




**Scope and Visibility of Variables:**

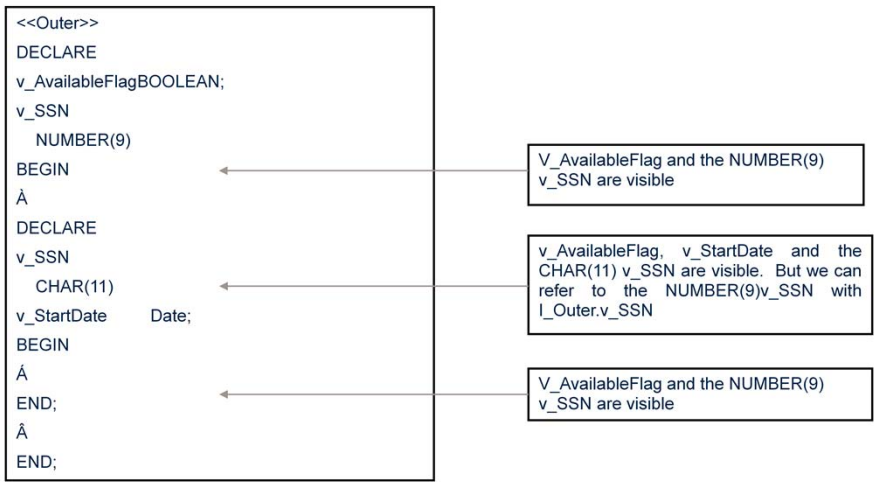
- References to an identifier are resolved according to its scope and visibility.
  - The scope of an identifier is that region of a program unit (block, subprogram, or package) from which you can reference the identifier.
  - An identifier is visible only in the regions from which you can reference the identifier using an unqualified name.
- Identifiers declared in a PL/SQL block are considered "local" to that "block" and "global" to all its "sub-blocks".
  - If a global identifier is re-declared in a sub-block, both identifiers remain in scope. However, the local identifier is visible within the sub-block only because you must use a qualified name to reference the global identifier.
- Although you cannot declare an identifier twice in the same block, you can declare the same identifier in two different blocks.
  - The two items represented by the identifier are "distinct", and any change in one does not affect the other. However, a block cannot reference identifiers declared in other blocks at the same level because those identifiers are neither local nor global to the block.


1.4: Scope and Visibility of variables
Scope and Visibility of Variables
Visibility of Variables:
The visibility of a variable is the portion of the program, where the variable can be accessed without having to qualify the reference. The visibility is always within the scope, it is not visible.
Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING
Copyright © Capgemini 2015. All Rights Reserved 26

1.4: Scope and Visibility of variables

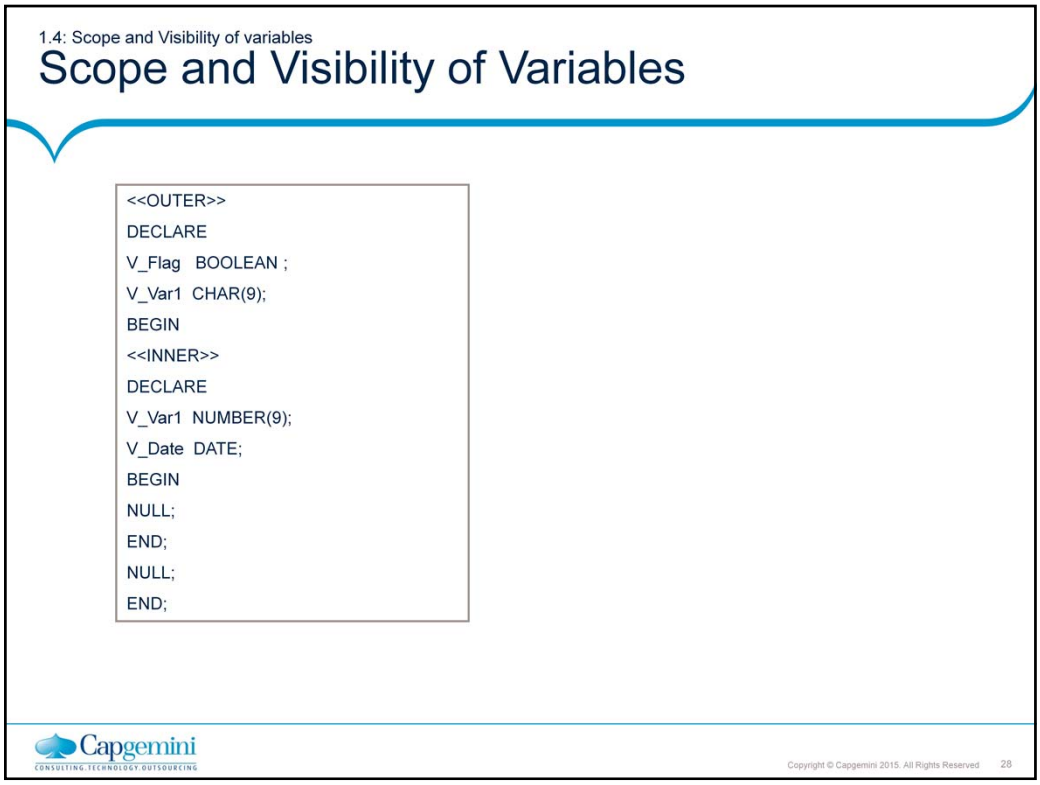
## Scope and Visibility of Variables

- Pictorial representation of visibility of a variable:




Capgemini CONSULTING.TECHNOLOGY.OUTSOURCING

 27


1.4: Scope and Visibility of variables
Scope and Visibility of Variables
<<OUTER>>
DECLARE
V_Flag BOOLEAN ;
V_Var1 CHAR(9);
BEGIN
<<INNER>>
DECLARE
V_Var1 NUMBER(9);
V_Date DATE;
BEGIN
NULL;
END;
NULL;
END;
Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING
Copyright © Capgemini 2015. All Rights Reserved
28

1.5: SQL in PL/SQL

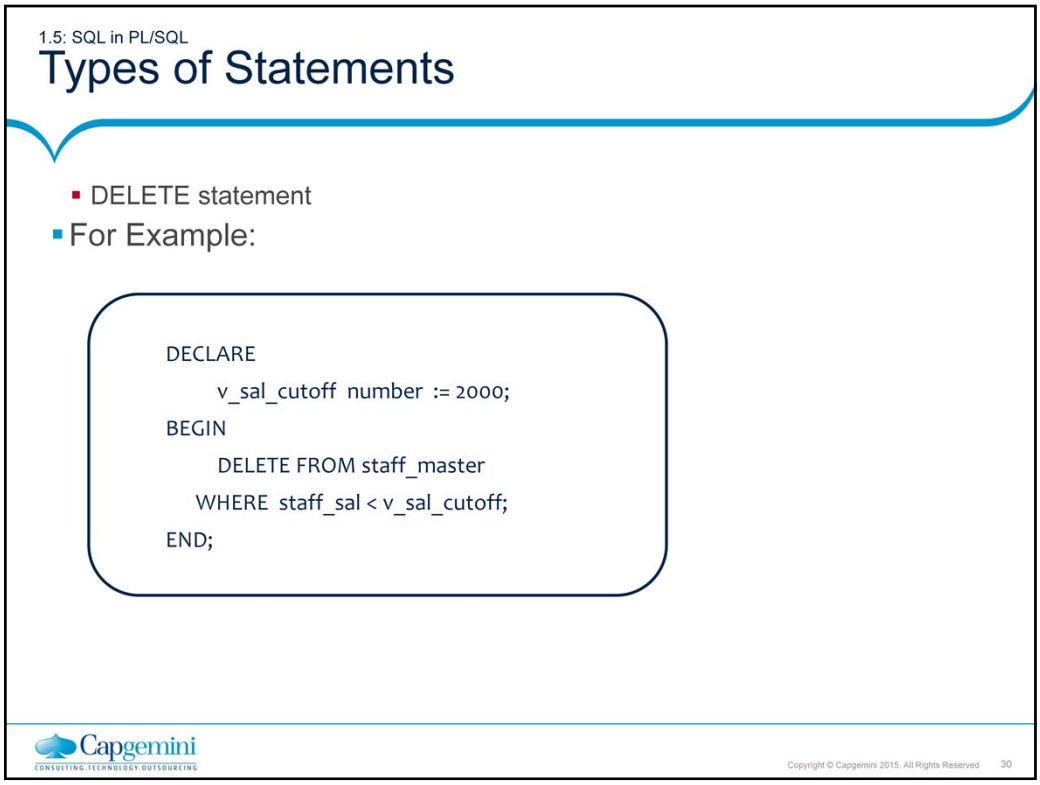
## Types of Statements

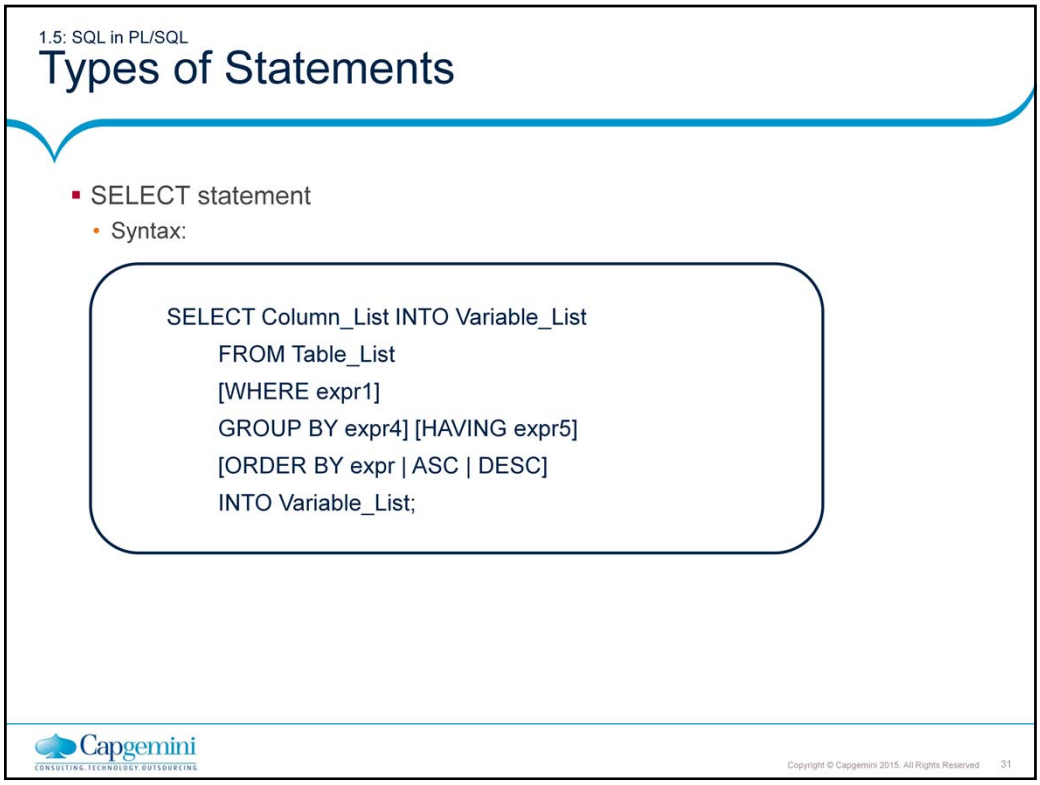
- Given below are some of the SQL statements that are used in PL/SQL:
  - INSERT statement
    - The syntax for the INSERT statement remains the same as in SQL-INSERT.
    - For example:

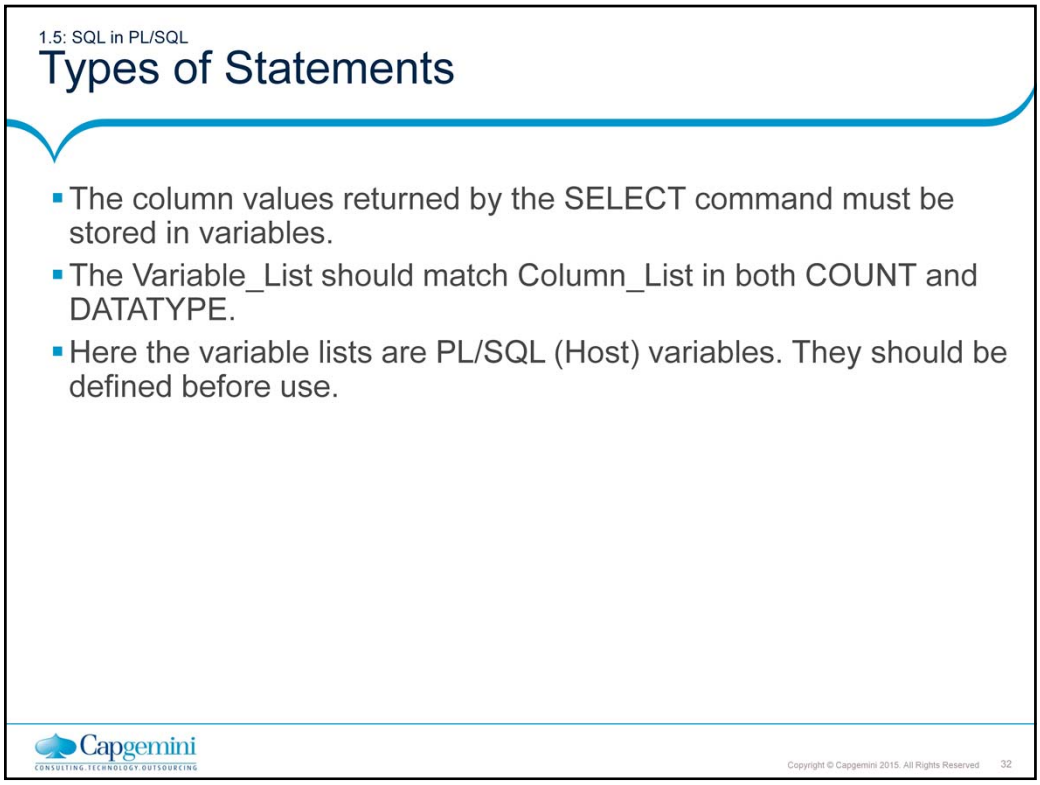
```
DECLARE
      v_dname varchar2(15) := 'Accounts';
BEGIN
       INSERT into department_master
      VALUES (50, v_dname);
END;
```

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 29


1.5: SQL in PL/SQL
Types of Statements
DELETE statement
For Example:
DECLARE
v_sal_cutoff number := 2000;
BEGIN
DELETE FROM staff_master
WHERE staff_sal < v_sal_cutoff;
END;
Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING
Copyright © Capgemini 2015. All Rights Reserved
30

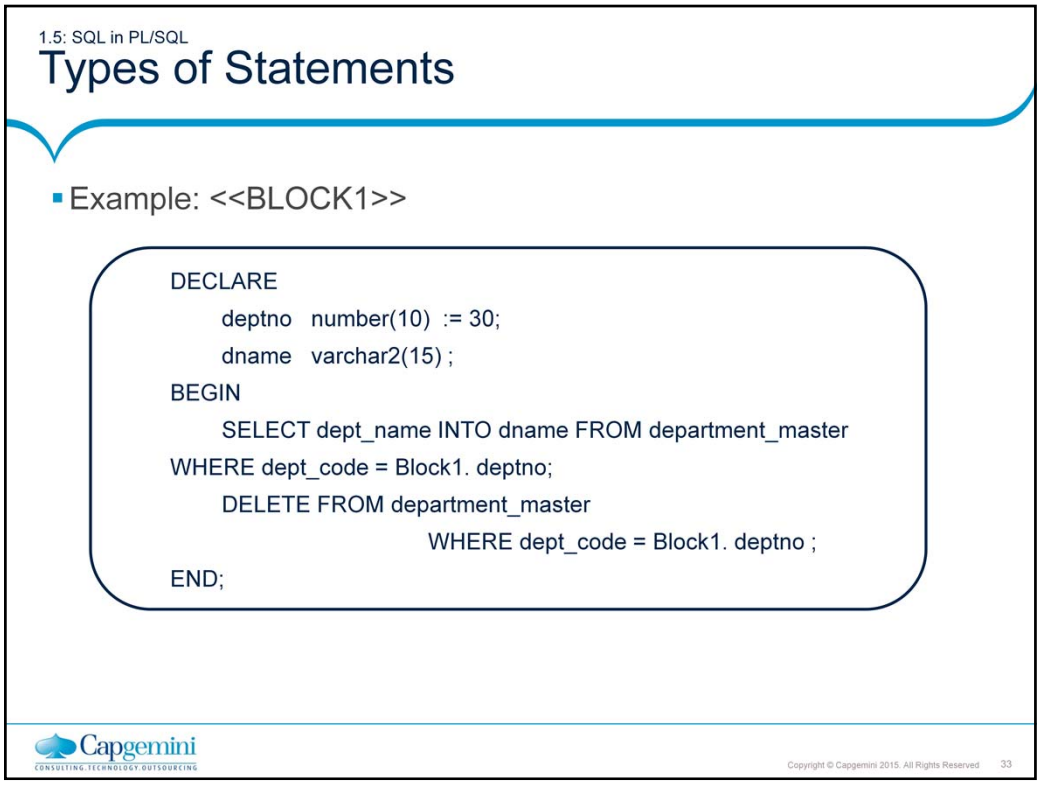

1.5: SQL in PL/SQL
Types of Statements
SELECT statement
Syntax:
SELECT Column_List INTO Variable_List
FROM Table_List
[WHERE expr1]
GROUP BY expr4] [HAVING expr5]
[ORDER BY expr | ASC | DESC]
INTO Variable_List;
Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING
Copyright © Capgemini 2015. All Rights Reserved
31




**SELECT Statement:**

**Note:**

- The SELECT clause is used if the selected row must be modified through a DELETE or UPDATE command.



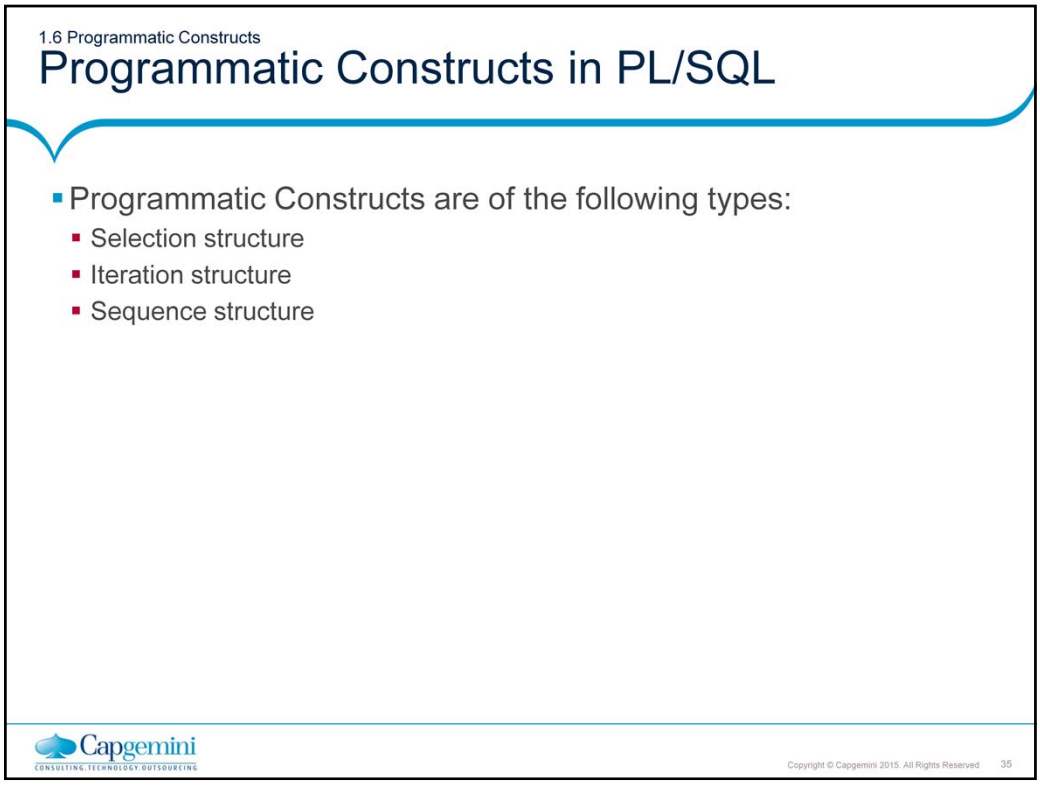

**SELECT statement (contd.):**

**SELECT statement (contd.):**

**More examples**

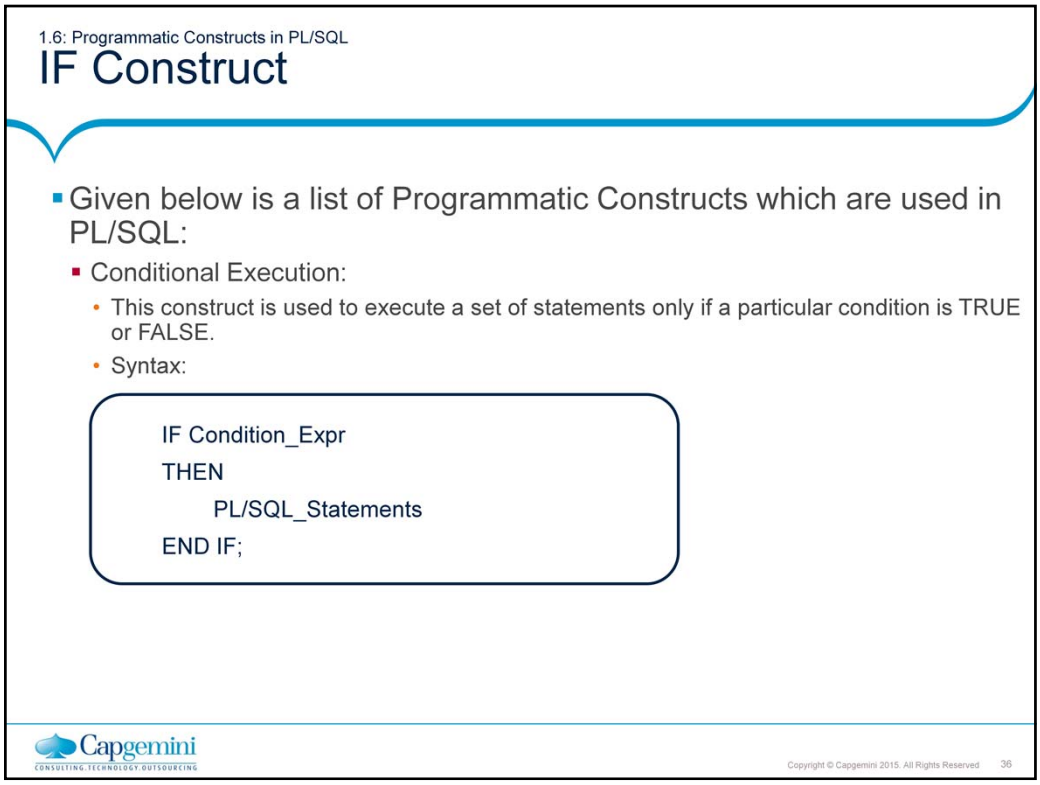
- Here the SELECT statement will select names of all the departments and not only deptno. 30.
- The DELETE statement will delete all the employees.
  - This happens because when the PL/SQL engine sees a condition expr1 = expr2, the expr1 and expr2 are first checked to see whether they match the database columns first and then the PL/SQL variables.
  - So in the above example, where you see deptno = deptno, both are treated as database columns, and the condition will become TRUE for every row of the table.
- If a block has a label, then variables with same names as database columns can be used by using <<blockname>>. Variable_Name notation.
- It is not a good programming practice to use same names for PL/SQL variables and database columns.

```
DECLARE
dept_code          number(10)  := 30;
v_dname            varchar2(15) ;
BEGIN
SELECT dept_name INTO v_dname FROM
department_master  WHERE dept_code=dept_code
DELETE FROM department_master
        WHERE dept_code = dept_code ;
END;
```




**Programming Constructs:**

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

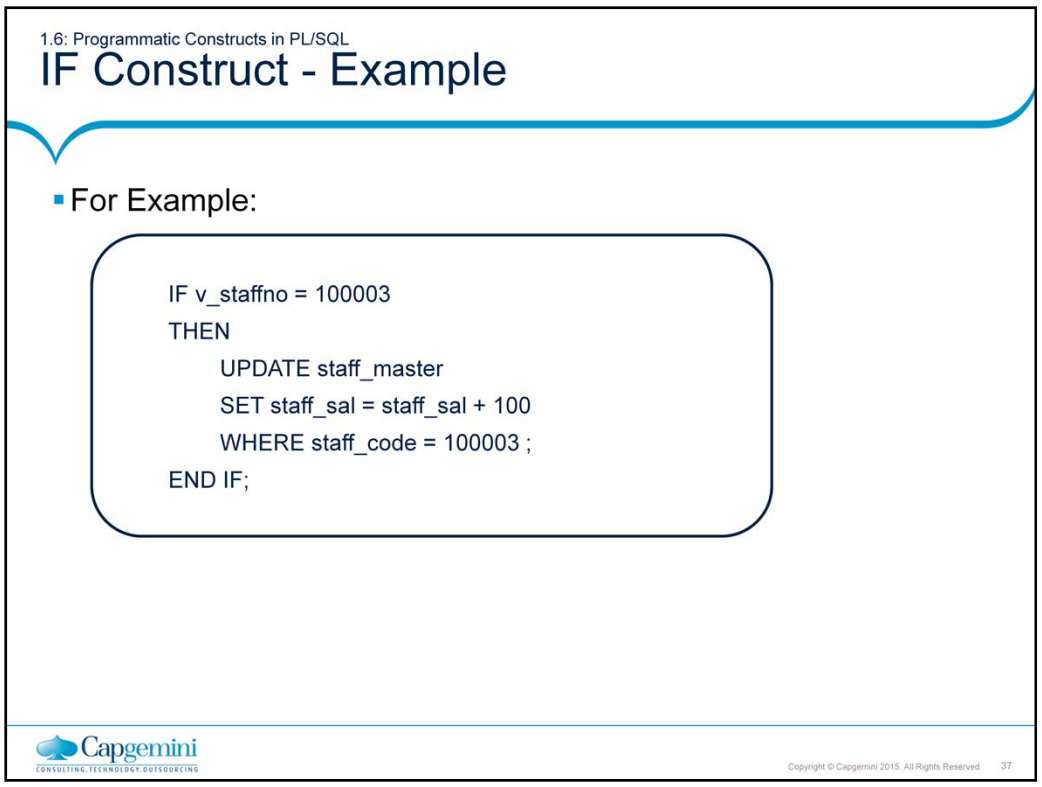



**Programmatic Constructs (contd.)**

**Conditional Execution:**

- Conditional execution is of the following type:
  - IF-THEN-END IF
  - IF-THEN-ELSE-END IF
  - IF-THEN-ELSIF-END IF
- Conditional Execution construct is used to execute a set of statements only if a particular condition is TRUE or FALSE.

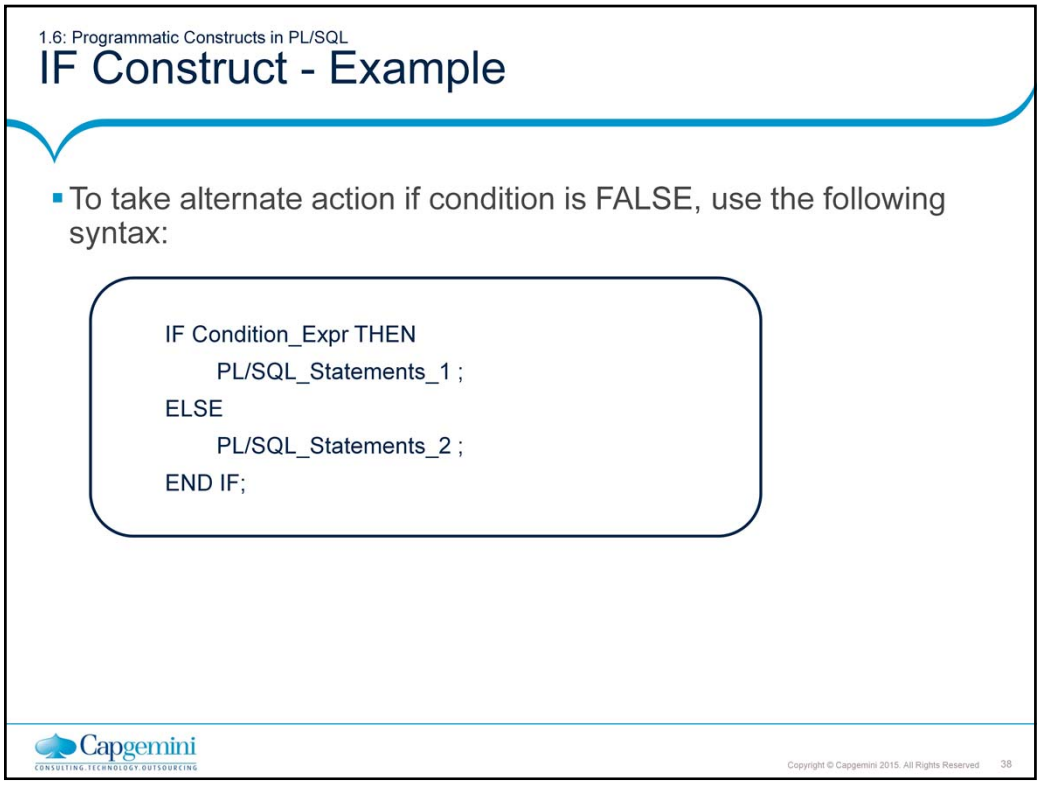
contd.




**Programmatic Constructs (contd.)**

**Conditional Execution (contd.):**

- As shown in the example in the slide, when the condition evaluates to TRUE, the PL/SQL statements are executed, otherwise the statement following END IF is executed.
- UPDATE statement is executed only if value of v_staffno variable equals 100003.
- PL/SQL allows many variations for the IF – END IF construct.

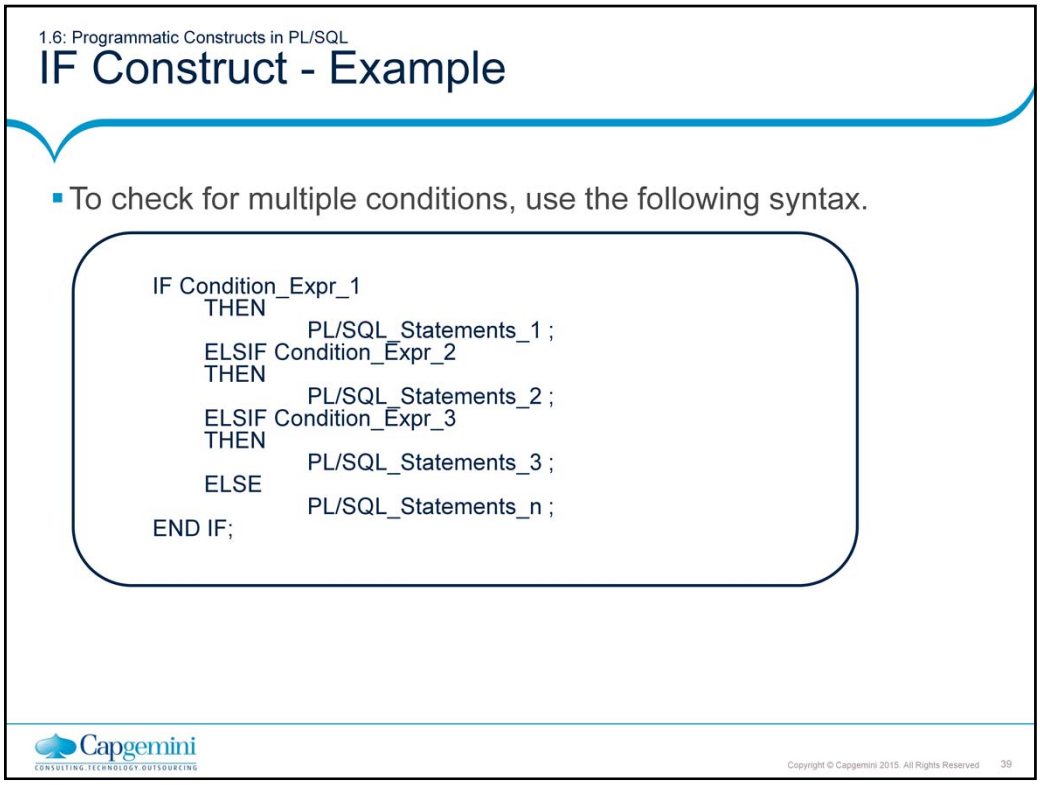



**Programmatic Constructs (contd.)**

**Conditional Execution (contd.):**

**Note:**

- When the condition evaluates to TRUE, the PL/SQL_Statements_1 is executed, otherwise PL/SQL_Statements_2 is executed.
- The above syntax checks **only one** condition, namely Condition_Expr.




**Programmatic Constructs (contd.)**
**Conditional Execution (contd.):**

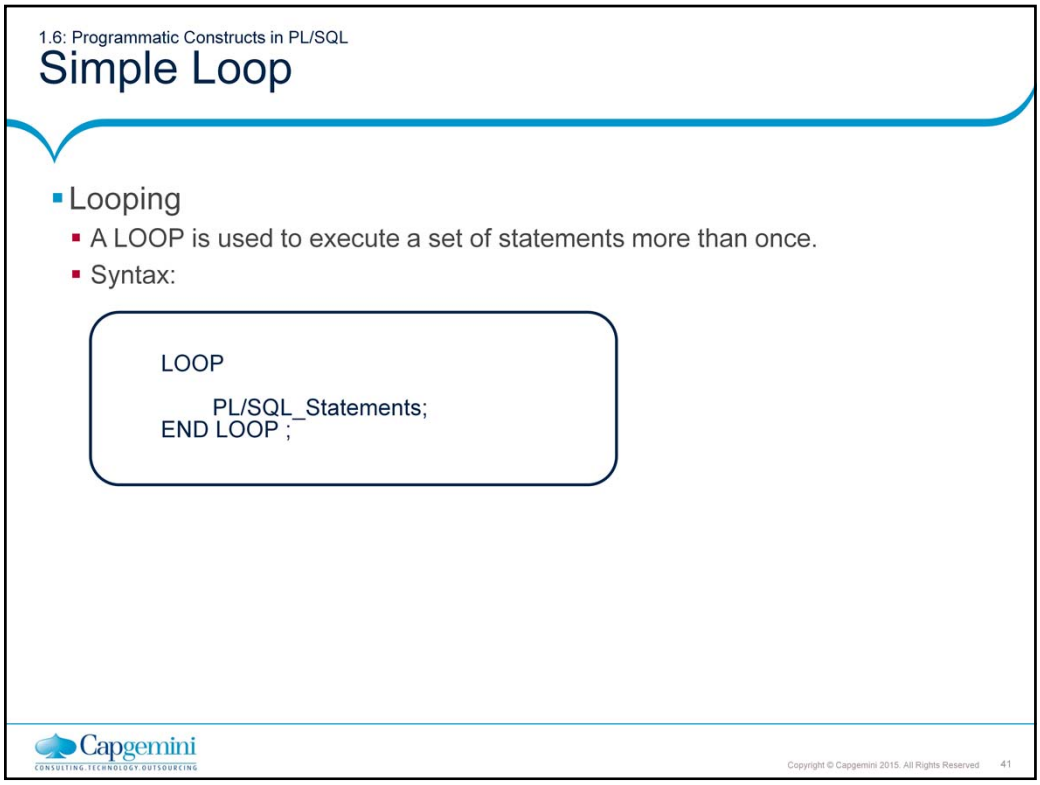
```
DECLARE
    D VARCHAR2(3): = TO_CHAR(SYSDATE, 'DY')
BEGIN
    IF   D= 'SAT'    THEN
        DBMS_OUTPUT.PUT_LINE('ENJOY YOUR
WEEKEND');
    ELSIF  D= 'SUN'    THEN
        DBMS_OUTPUT.PUT_LINE('ENJOY YOUR
WEEKEND');
    ELSE
        DBMS_OUTPUT.PUT_LINE('HAVE A NICE
DAY');
    END IF;
END;
```

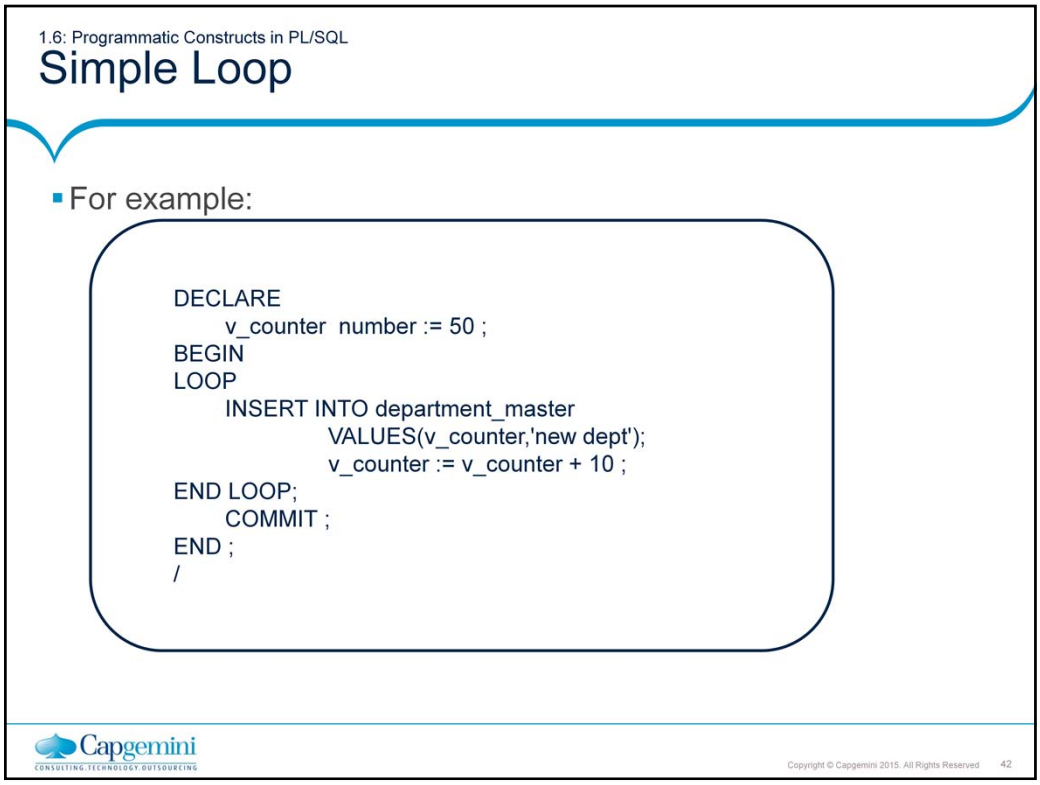
**Programmatic Constructs (contd.)**

**Conditional Execution (contd.):**

- As every condition must have at least one statement, NULL statement can be used as filler.
- NULL command does nothing.
- Sometimes NULL is used in a condition merely to indicate that such a condition has been taken into consideration, as well.
- Conditions for NULL are checked through IS NULL and IS NOT NULL predicates.

```
IF Condition_Expr_1 THEN

PL/SQL_Statements_1 ;
ELSIF Condition_Expr_2 THEN

PL/SQL_Statements_2 ;
ELSIF Condition_Expr_3 THEN
        Null;
END IF;
```


1.6: Programmatic Constructs in PL/SQL
Simple Loop
Looping
A LOOP is used to execute a set of statements more than once.
Syntax:
LOOP
PL/SQL_Statements;
END LOOP ;
Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING
Copyright © Capgemini 2015. All Rights Reserved
41




**Programmatic Constructs (contd.)**

**Looping**

- The example shown in the slide is an endless loop.
- When LOOP ENDLOOP is used in the above format, then an exit path must necessarily be provided. This is discussed in the following slide.

1.6: Programmatic Constructs in PL/SQL

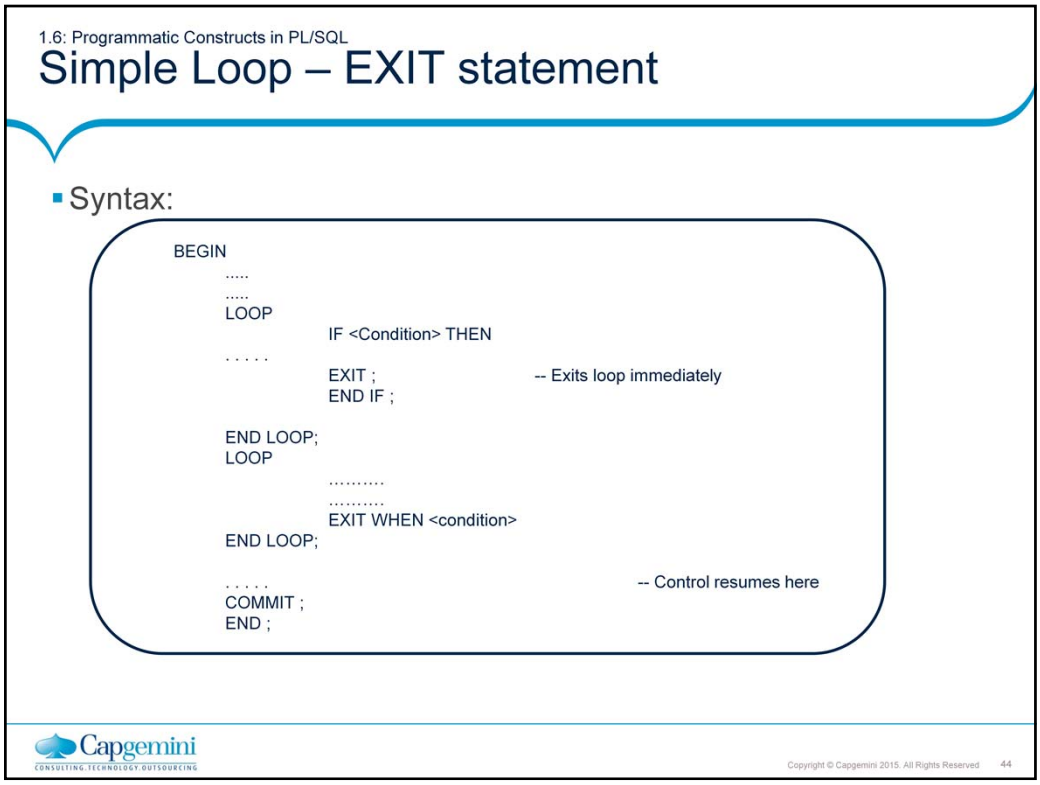
## Simple Loop – EXIT statement

- EXIT
  - Exit path is provided by using EXIT or EXIT WHEN commands.
  - EXIT is an unconditional exit. Control is transferred to the statement following END LOOP, when the execution flow reaches the EXIT statement.

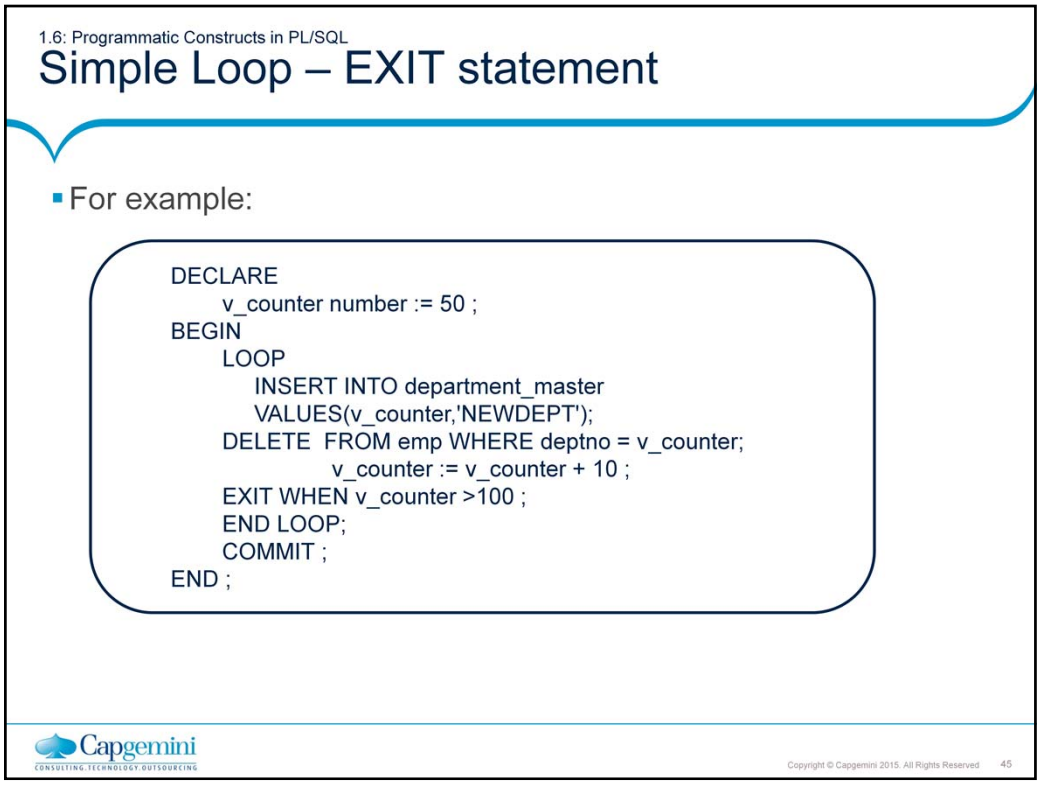
contd.

Capgemini
CONSULTING. TECHNOLOGY. OUTSOURCING

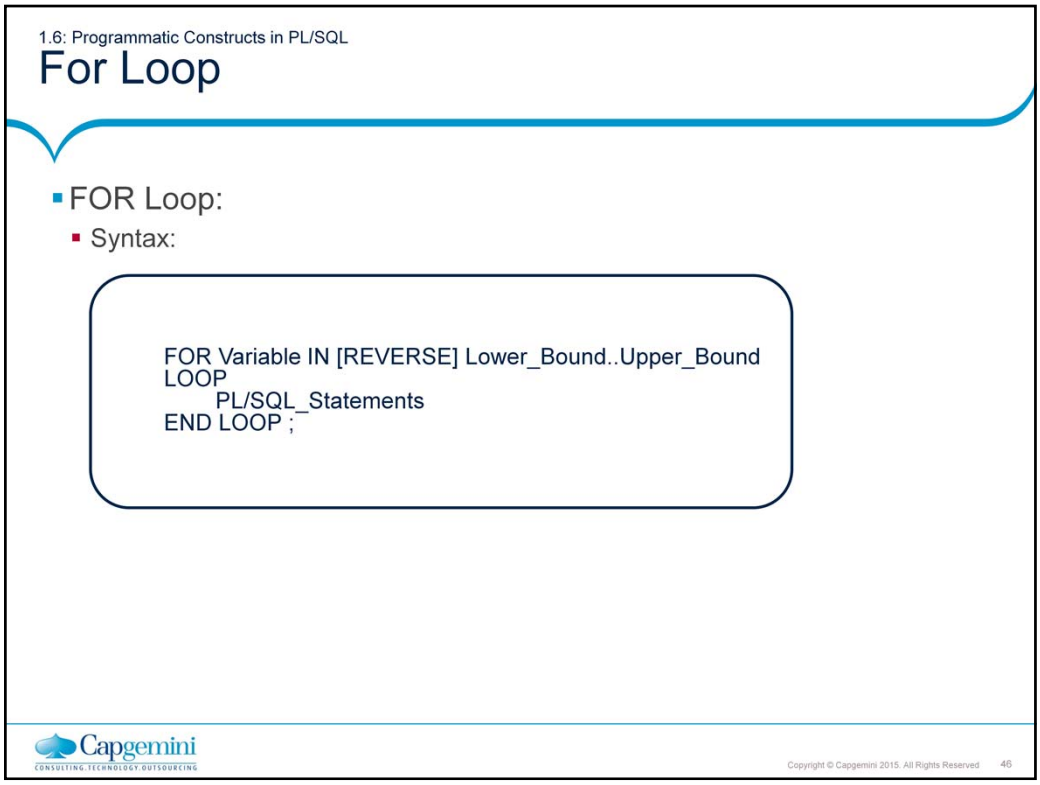
 43




**Note:**
EXIT WHEN is used for conditional exit out of the loop.



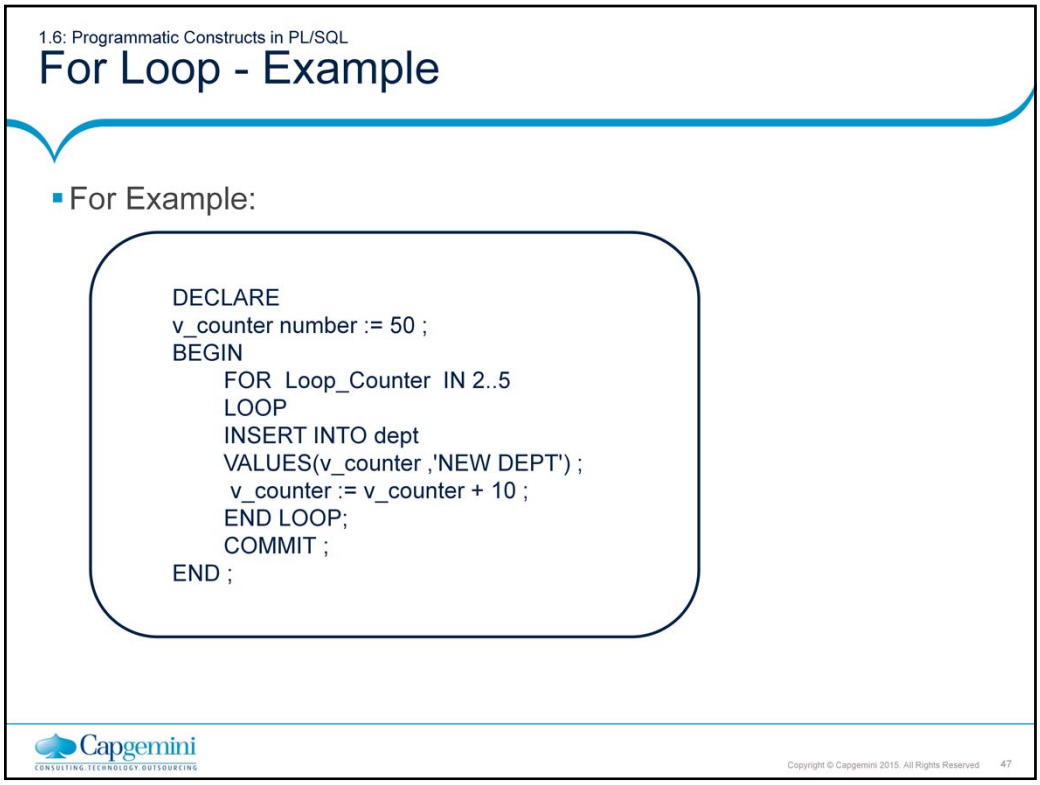

**Note:**
LOOP.. END LOOP can be used in conjunction with FOR and WHILE for better control on looping.




**Programmatic Constructs (contd.)**

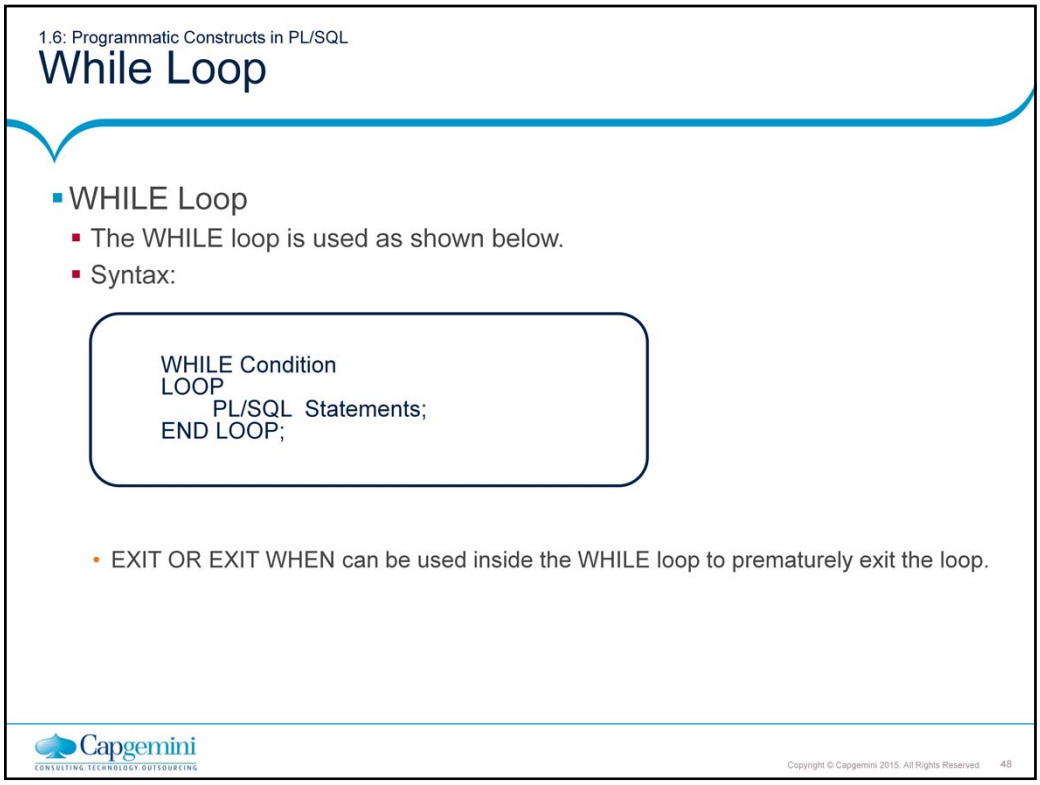
**FOR Loop:**

- FOR loop is used for executing the loop a fixed number of times. The number of times the loop will execute equals the following:
  - Upper_Bound - Lower_Bound + 1.
- Upper_Bound and Lower_Bound must be integers.
- Upper_Bound must be equal to or greater than Lower_Bound.
- Variables in FOR loop need not be explicitly declared.
  - Variables take values starting at a Lower_Bound and ending at a Upper_Bound.
  - The variable value is incremented by 1, every time the loop reaches the bottom.
  - When the variable value becomes equal to the Upper_Bound, then the loop executes and exits.
- When REVERSE is used, then the variable takes values starting at Upper_Bound and ending at Lower_Bound.
- Value of the variable is decremented each time the loop reaches the bottom.




**Programmatic Constructs (contd.)**

- In the example in the above slide, the loop will be executed (5 - 2 + 1) = 4 times.
- A Loop_Counter variable can also be used inside the loop, if required.
- Lower_Bound and/or Upper_Bound can be integer expressions, as well.

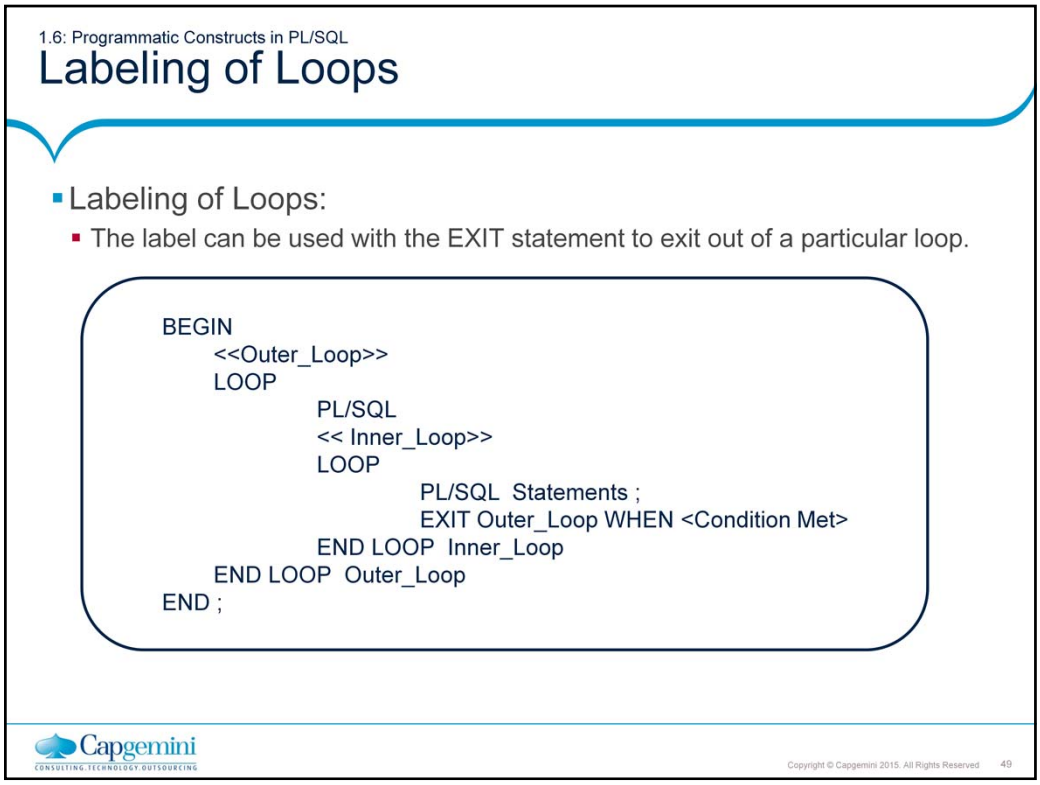



**Programmatic Constructs (contd.)**

**WHILE Loop:**

**Example:**

```
DECLARE
     ctr number := 1;
BEGIN
     WHILE ctr <= 10
     LOOP
     dbms_output.put_line(ctr);
          ctr := ctr+1;
     END LOOP;
END;
/
```




**Programmatic Constructs (contd.)**

**Labeling of Loops:**

- Loops themselves can be labeled as in the case of blocks.
- The label can be used with the EXIT statement to exit out of a particular loop.

## Summary

- In this lesson, you have learnt:
  - PL/SQL is a procedural extension to SQL.
  - PL/SQL exhibits a block structure, different block types being: Anonymous, Procedure, and Function.
  - While declaring variables in PL/SQL:
    - declare and initialize variables within the declaration section
    - assign new values to variables within the executable section


Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 50

# Summary

In this lesson, you have learnt:

- PL/SQL is a procedural extension to SQL.
- PL/SQL exhibits a block structure, different block types being: Anonymous, Procedure, and Function.
- While declaring variables in PL/SQL:
  - declare and initialize variables within the declaration section
  - assign new values to variables within the executable section


Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 51

## Summary

- Different types of PL/SQL Variables are: Scalar, Composite, Reference, LOB
- Scope of a variable: It is the portion of a program in which the variable can be accessed.
- Visibility of a variable: It is the portion of the program, where the variable can be accessed without having to qualify the reference.
- Different programmatic constructs in PL/SQL are Selection structure, Iteration structure, Sequence structure


Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 52

## Review & Question

- Question 1: A record is a collection of individual fields that represents a row in the table.
  - True/ False
- Question 2: %ROWTYPE is used to declare a variable with the same datatype as a column of a specific table.
  - True / False
- Question 3: While using FOR loop, Upper_Bound, and Lower_Bound must be integers.
  - True / False

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 53

# Oracle Basics (PL/SQL)

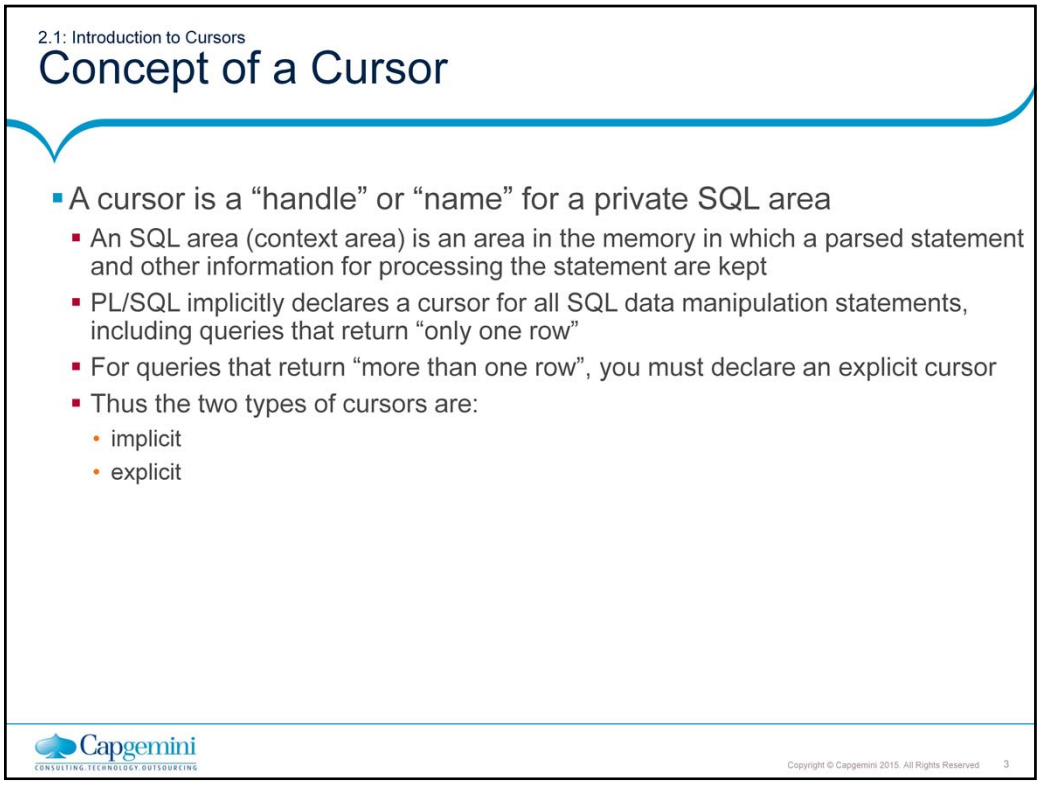
Lesson 02 Introduction to Cursors

## Lesson Objectives

- To understand the following topics:
  - Introduction to Cursors
  - Implicit Cursors
  - Explicit Cursors
  - Cursor attributes
  - Processing Implicit Cursors and Explicit Cursors

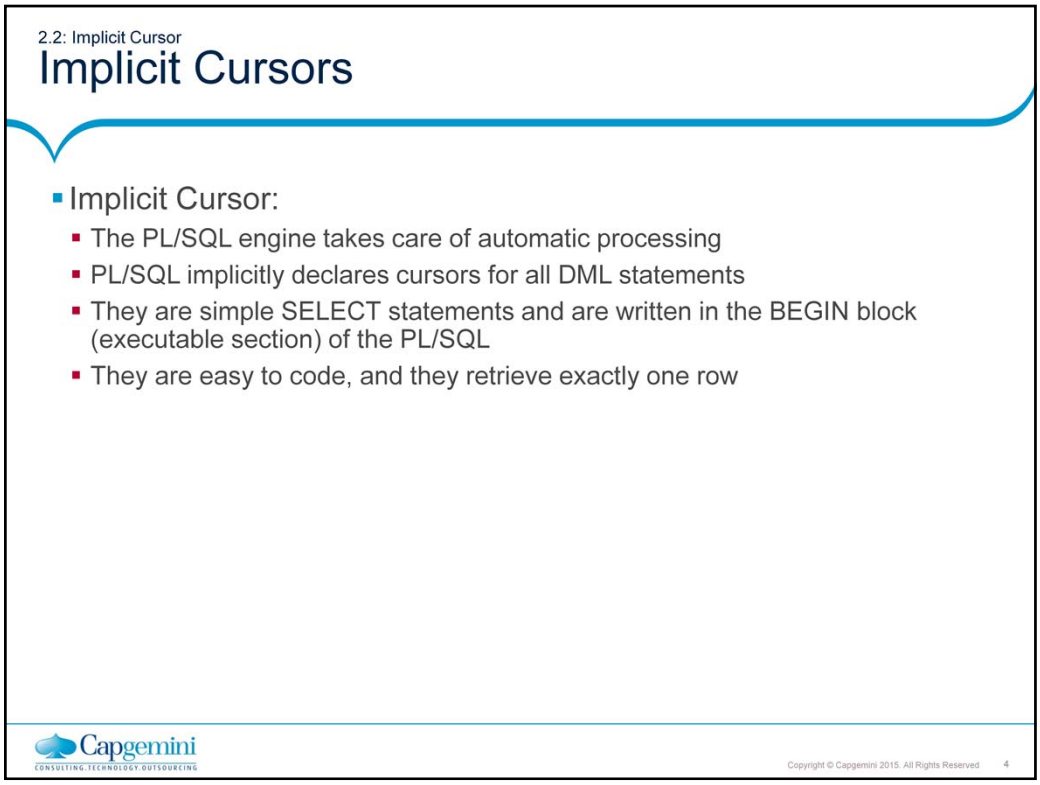
Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

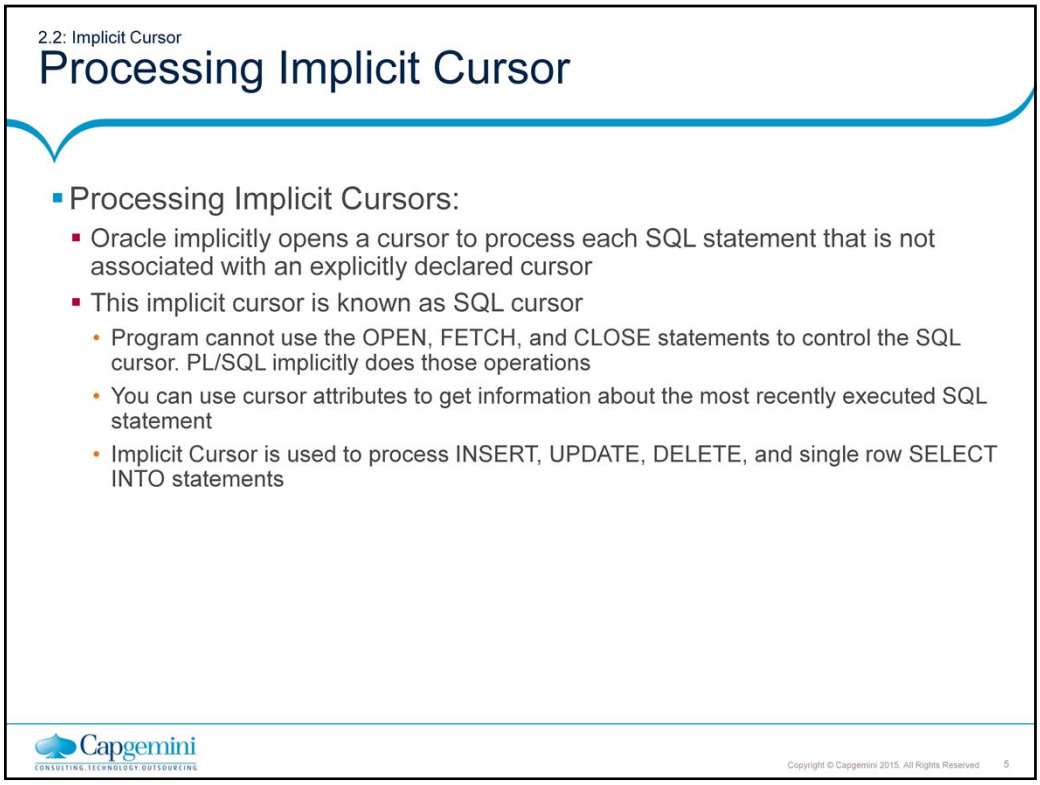
 2




**Introduction to Cursors:**

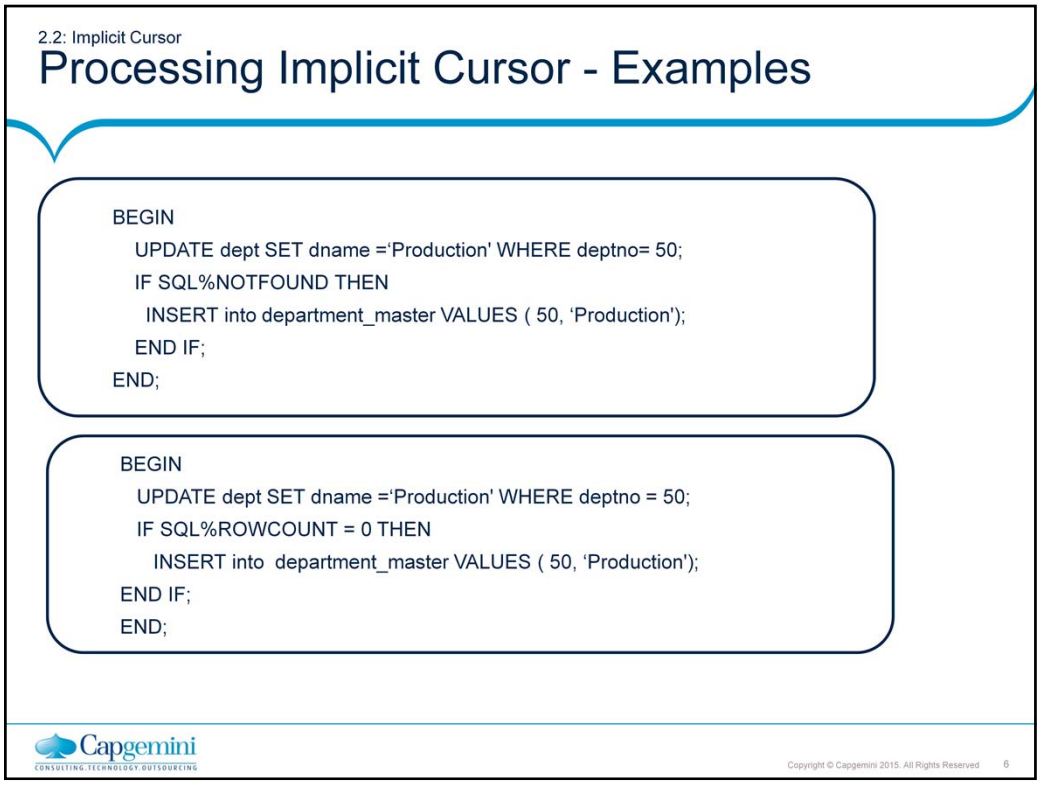
- ORACLE allocates memory on the Oracle server to process SQL statements. It is called as “**context area**”. Context area stores information like number of rows processed, the set of rows returned by a query, etc.
- A Cursor is a “handle” or “pointer” to the context area. Using this cursor the PL/SQL program can control the context area, and thereby access the information stored in it.
- There are two types of cursors - “explicit” and “implicit”.
  - In an explicit cursor, a cursor name is explicitly assigned to a SELECT statement through CURSOR IS statement.
  - An implicit cursor is used for all other SQL statements.
- Processing an explicit cursor involves four steps. In case of implicit cursors, the PL/SQL engine automatically takes care of these four steps.


2.2: Implicit Cursor
Implicit Cursors
Implicit Cursor:
The PL/SQL engine takes care of automatic processing
PL/SQL implicitly declares cursors for all DML statements
They are simple SELECT statements and are written in the BEGIN block (executable section) of the PL/SQL
They are easy to code, and they retrieve exactly one row
Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING
Copyright © Capgemini 2015. All Rights Reserved 4




**Processing Implicit Cursors:**

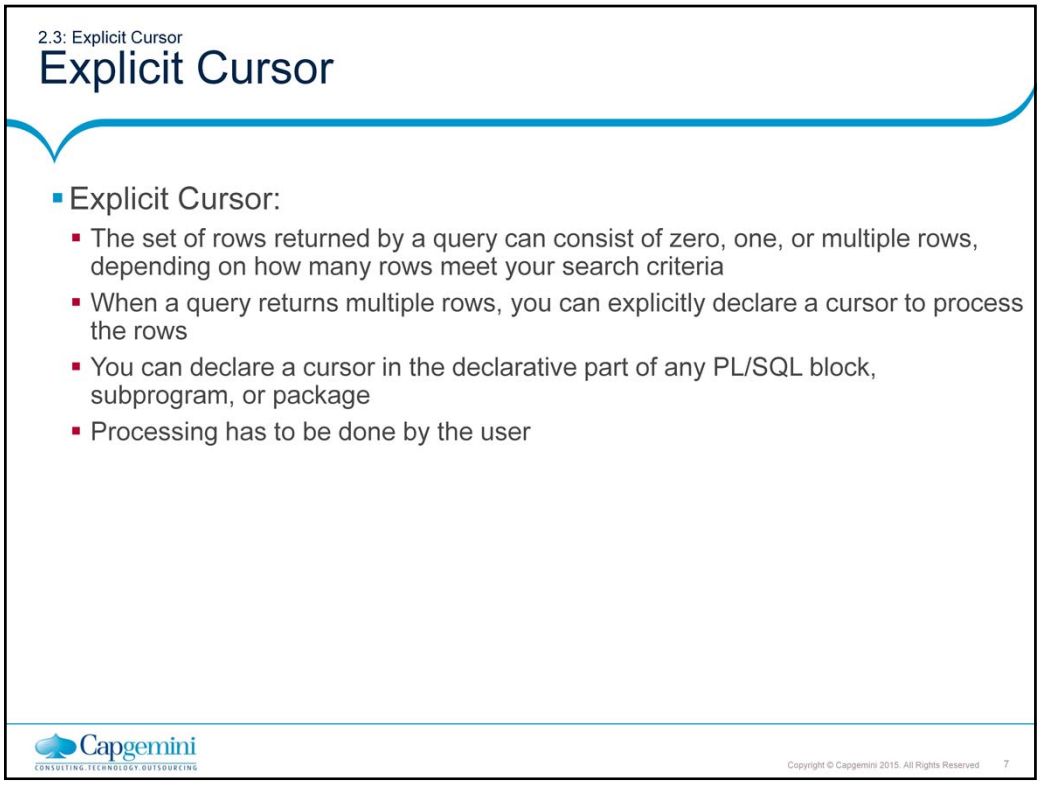
- All SQL statements are executed inside a context area and have a cursor, which points to that context area. This implicit cursor is known as SQL cursor.
- Implicit SQL cursor is not opened or closed by the program. PL/SQL implicitly opens the cursor, processes the SQL statement, and closes the cursor.
- Implicit cursor is used to process INSERT, UPDATE, DELETE, and single row SELECT INTO statements.
- The cursor attributes can be applied to the SQL cursor.




**Processing Implicit Cursors:**

**Note:**

- SQL%NOTFOUND should not be used with SELECT INTO …. Statement.
- This is because SELECT INTO…. Statement will raise an ORACLE error if no rows are selected, and
  - control will pass to exception * section (discussed later), and
  - SQL%NOTFOUND statement will not be executed at all
- The slide shows two code snippets using Cursor attributes SQL%NOTFOUND and SQL%ROWCOUNT respectively.




**Explicit Cursor:**

- When you need precise control over query processing, you can explicitly declare a cursor in the declarative part of any PL/SQL block, subprogram, or package.
- This technique requires more code than other techniques such as the implicit cursor FOR loop. But it is beneficial in terms of flexibility. You can:
  - ➢Process several queries in parallel by declaring and opening multiple cursors.
  - ➢Process multiple rows in a single loop iteration, skip rows, or split the processing into more than one loop.

**Processing Explicit Cursors:**

**Overview of Explicit Cursors**

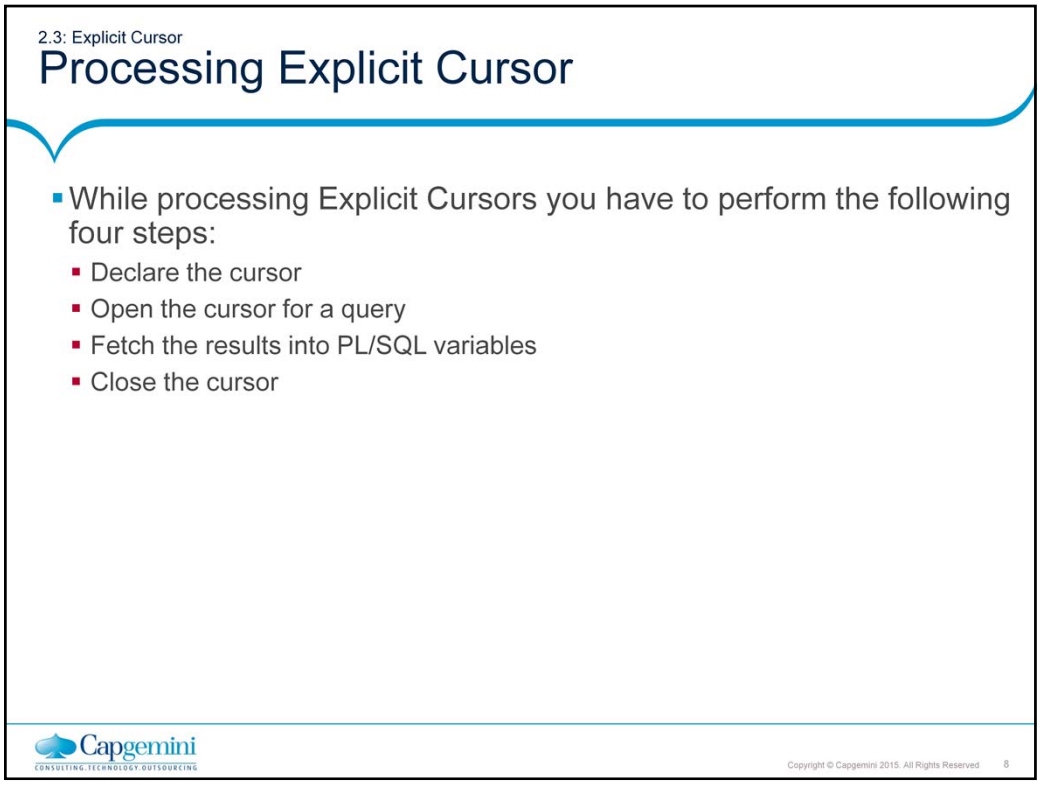
•You use three commands to control a cursor: OPEN, FETCH, and CLOSE.

•First, you initialize the cursor with the OPEN statement, which identifies the result set. Then, you can execute FETCH repeatedly until all rows have been retrieved, or you can use the BULK COLLECT clause to fetch all rows at once. When the last row has been processed, you release the cursor with the CLOSE statement.

**Declaring a Cursor**

You must declare a cursor before referencing it in other statements. You give the cursor a name and associate it with a specific query. You can optionally declare a return type for the cursor (such as table_name%ROWTYPE). You can optionally specify parameters that you use in the WHERE clause instead of referring to local variables. These parameters can have default values.

contd.



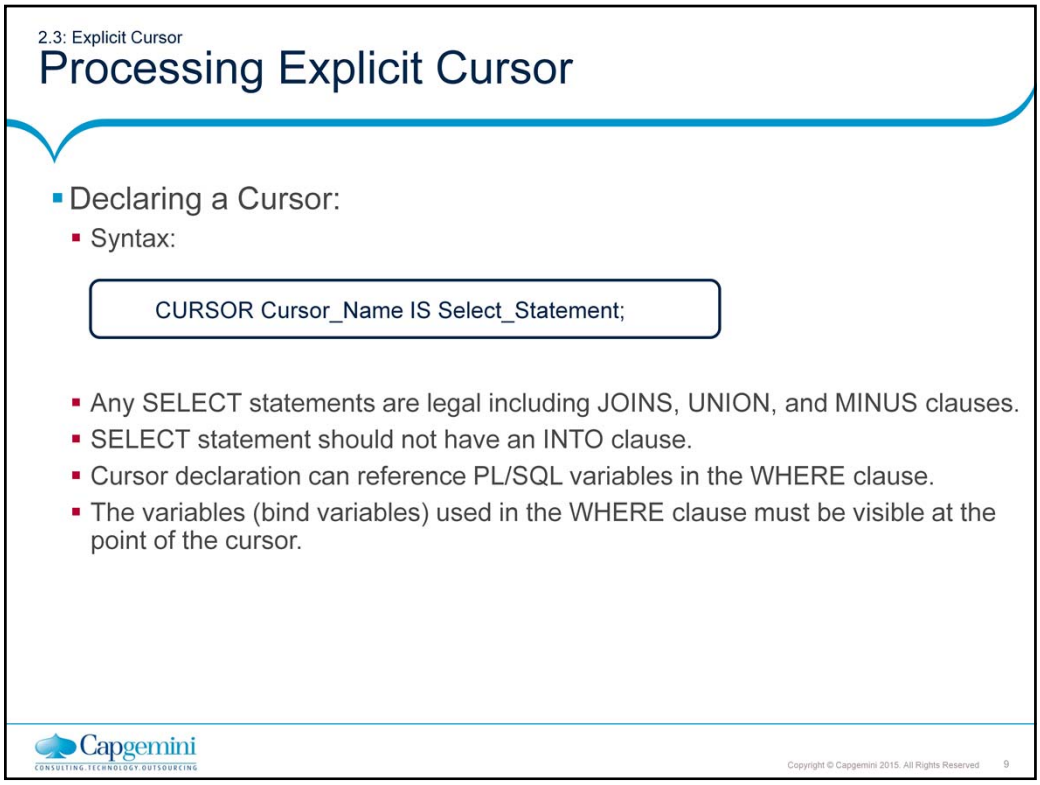
**Processing Explicit Cursors:**

**For example:** You might declare cursors like the one given below:

```
DECLARE
  CURSOR c1 IS SELECT
empno, ename, job, sal
FROM emp
    WHERE sal > 2000;
  CURSOR c2 RETURN
dept%ROWTYPE IS
    SELECT * FROM dept
WHERE deptno = 10;
```

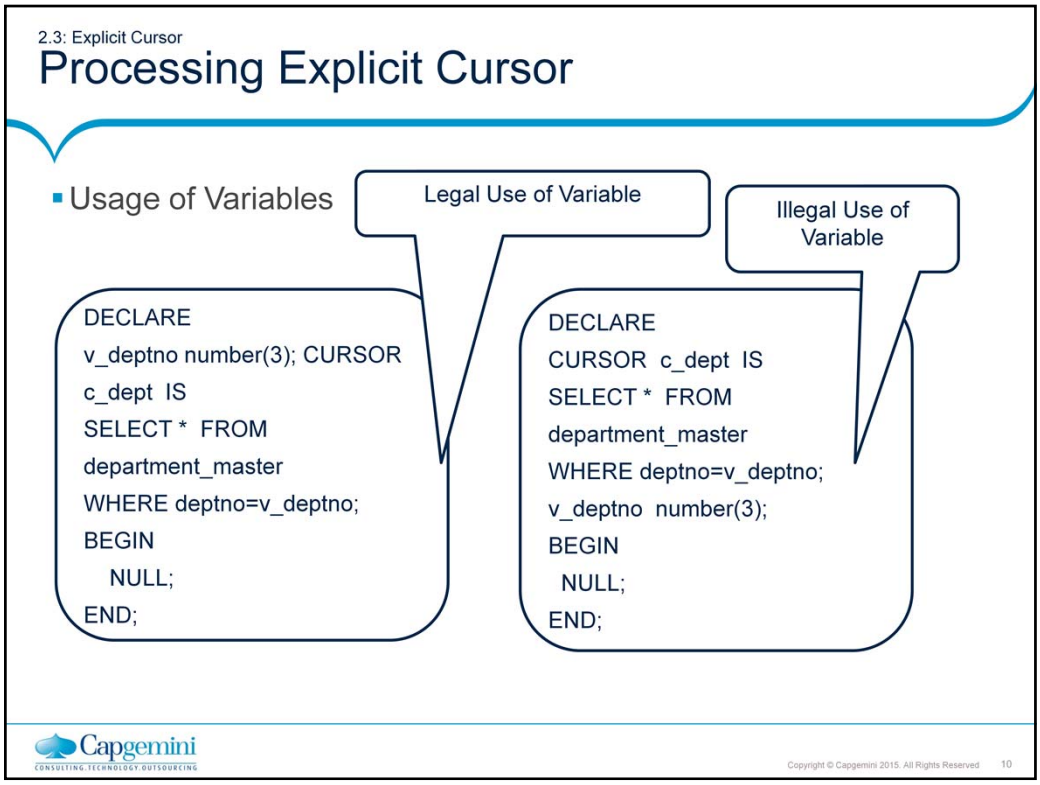
The cursor is not a PL/SQL variable; you cannot assign values to a cursor or use it in an expression. Cursors and variables follow the same scoping rules. Naming cursors after database tables is possible, however, it is not recommended.

contd.



**Processing Explicit Cursors:**

**Declaring a Cursor (contd.)**
contd.




**Processing Explicit Cursors: Declaring a Cursor:**

- The code snippets on the slide show the usage of variables in cursor declaration. You cannot use a variable before it has been declared. It will be illegal.

**Processing Explicit Cursors:**

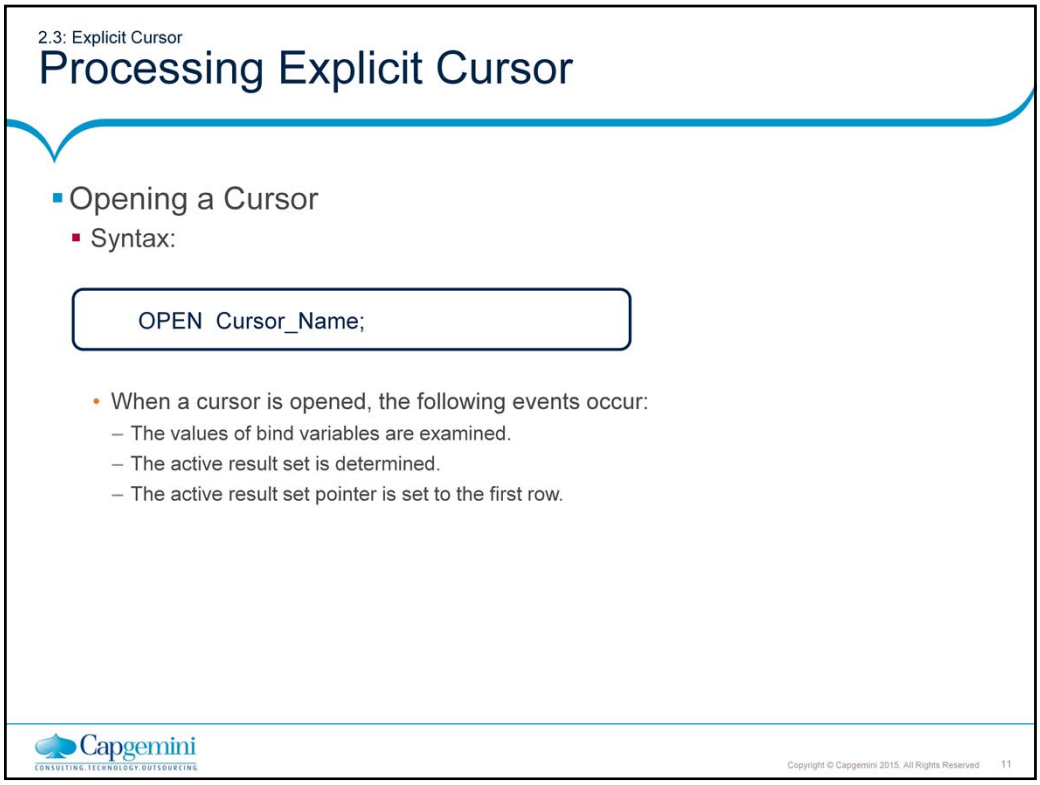
**Opening a Cursor**

•Opening the cursor executes the query and identifies the result set, which consists of all rows that meet the query search criteria.

•For cursors declared using the FOR UPDATE clause, the OPEN statement also locks those rows. An example of the OPEN statement follows:

```
DECLARE
  CURSOR c1 IS SELECT
ename, job FROM emp
WHERE sal < 3000;
  ...
BEGIN
  OPEN c1;
  ...
END;
```

Rows in the result set are retrieved by the FETCH statement, not when the OPEN statement is executed.




**Processing Explicit Cursors: Opening a Cursor:**

- When a Cursor is opened, the following events occur:
  1. The values of "bind variables" are examined.
  2. Based on the values of bind variables , the "active result set" is determined.
  3. The active result set pointer is set to the first row.
- "Bind variables" are evaluated only once at Cursor open time.
  - Changing the value of Bind variables after the Cursor is opened will not make any changes to the active result set.
  - The query will see changes made to the database that have been committed prior to the OPEN statement.
- You can open more than one Cursor at a time.

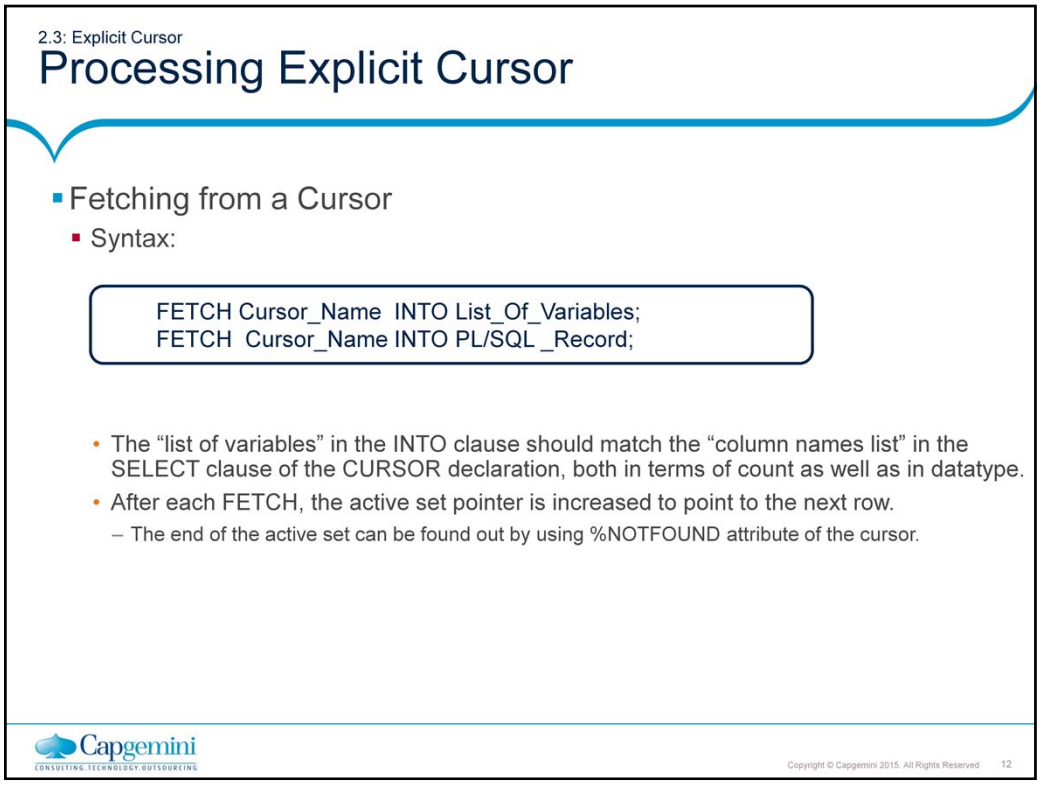
**Processing Explicit Cursors:**

**Fetching with a Cursor**
•Unless you use the BULK COLLECT clause the FETCH statement retrieves the rows in the result set one at a time. Each fetch retrieves the current row and advances the cursor to the next row in the result set.

•You can store each column in a separate variable, or store the entire row in a record that has the appropriate fields (usually declared using %ROWTYPE):

-- This cursor queries 3 columns.
-- Each column is fetched into a separate variable.
FETCH c1 INTO my_empno, my_ename, my_deptno;
-- This cursor was declared as SELECT * FROM employees.
-- An entire row is fetched into the my_employees record, which
-- is declared with the type employees%ROWTYPE.
FETCH c2 INTO my_employees;

contd.




**Processing Explicit Cursors: Fetching from Cursor:**

**Processing Explicit Cursors: Fetching with a Cursor (contd.)**

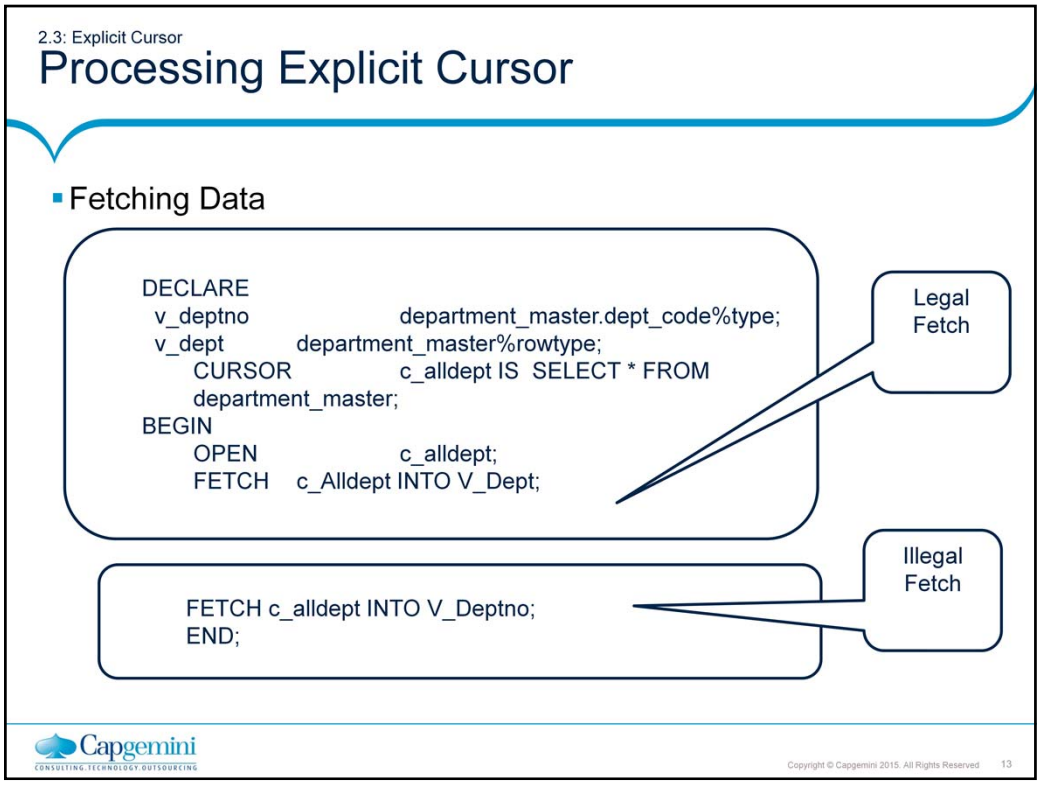
•Typically, you use the FETCH statement in the following way:

```
LOOP
  FETCH c1 INTO
my_record;
  EXIT WHEN
c1%NOTFOUND;
  -- process data record
END LOOP;
```

•The query can reference PL/SQL variables within its scope. Any variables in the query are evaluated only when the cursor is opened. In the following example, each retrieved salary is multiplied by 2, even though factor is incremented after every fetch:

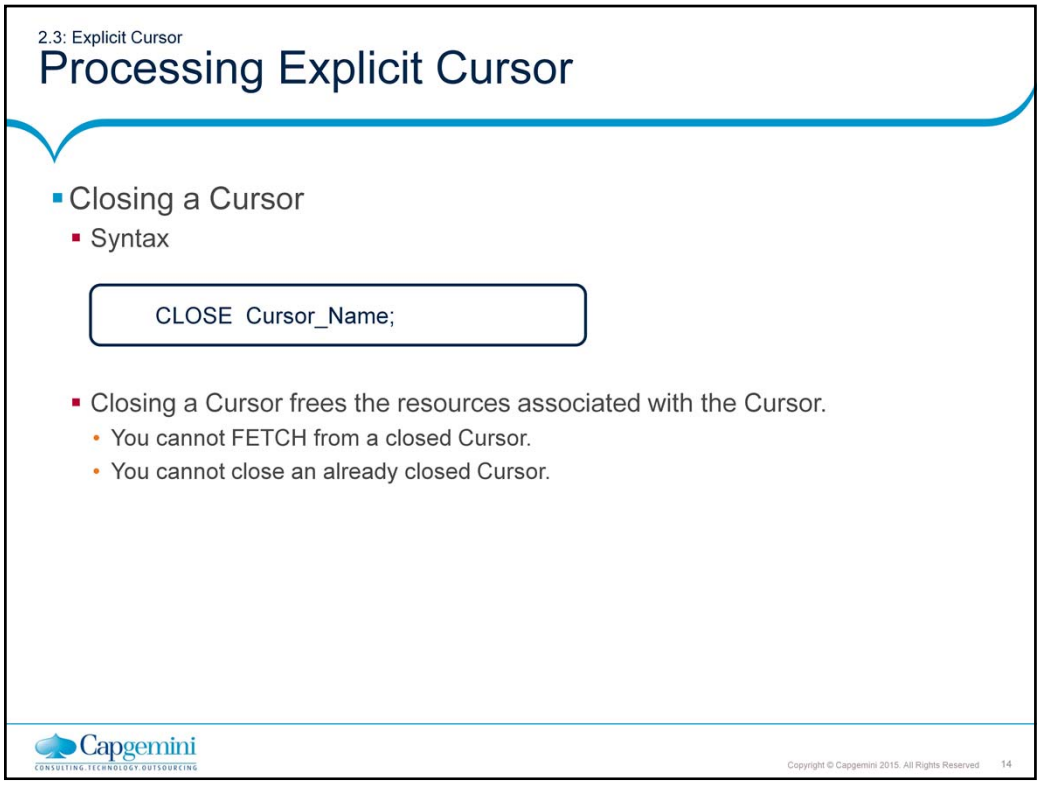
```
DECLARE
 my_sal
employees.salary%TYPE;
 my_job
employees.job_id%TYPE;
 factor INTEGER := 2;
 CURSOR c1 IS
  SELECT factor*salary
FROM employees WHERE
job_id = my_job;
BEGIN
  OPEN c1;  -- here factor
equals 2
  LOOP
    FETCH c1 INTO my_sal;
    EXIT WHEN
c1%NOTFOUND;
    factor := factor + 1;  --
does not affect FETCH
  END LOOP;
END;
/
```

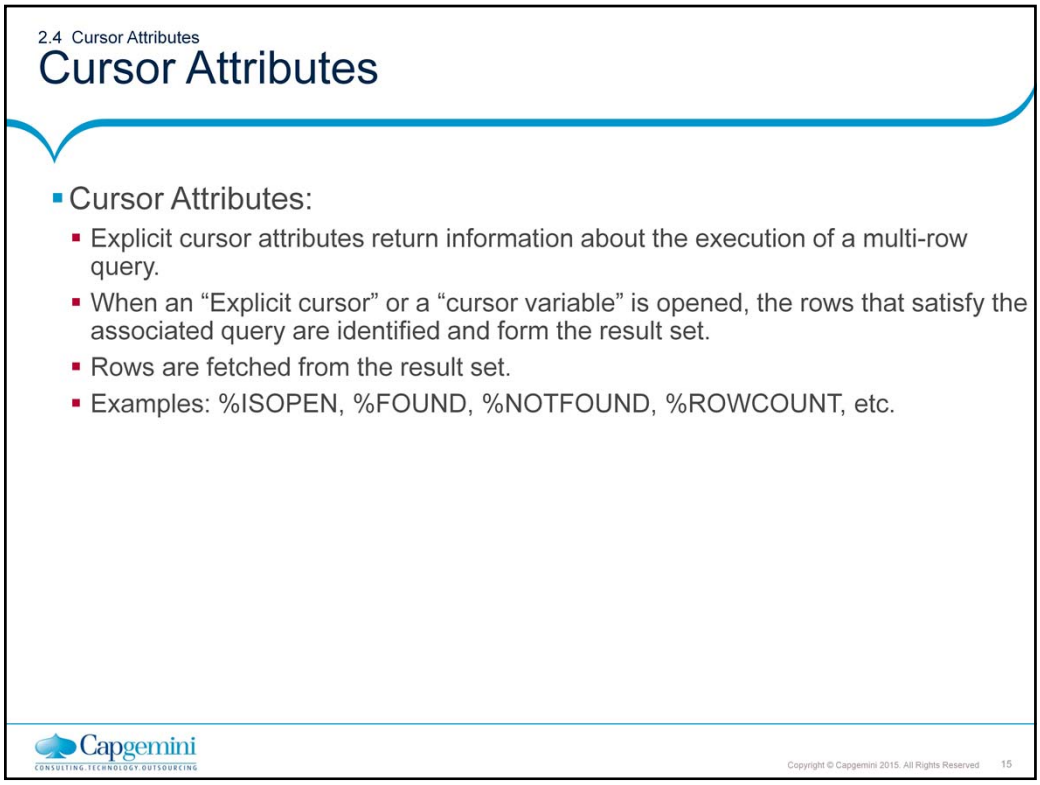
•you can use a different INTO list on separate fetches with the same cursor. Each fetch retrieves another row and assigns values to the target variables

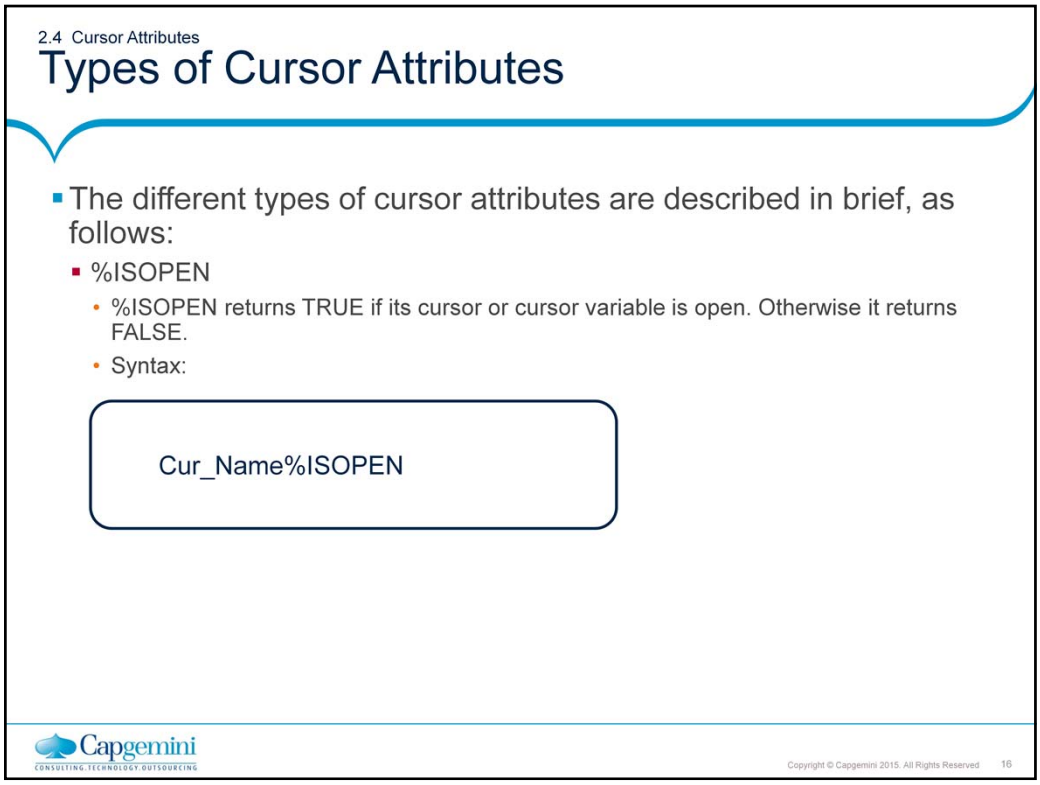



**Processing Explicit Cursors: Fetching from Cursor:**

- The code snippets on the slide shows example of fetching data from cursor. The second snippet FETCH is illegal since SELECT * selects all columns of the table, and there is only one variable in INTO list.
- For each column value returned by the query associated with the cursor, there must be a corresponding, type-compatible variable in the INTO list.
- To change the result set or the values of variables in the query, you must close and reopen the cursor with the input variables set to their new values.

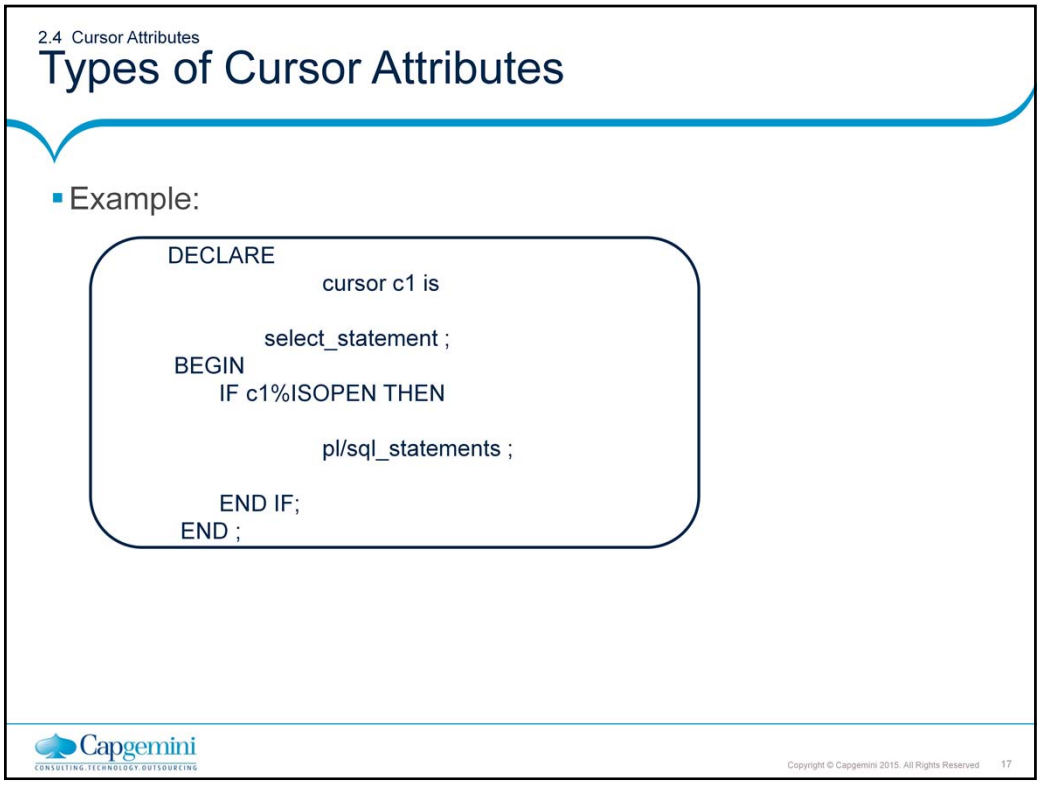

2.3: Explicit Cursor
Processing Explicit Cursor
Closing a Cursor
Syntax
CLOSE Cursor_Name;
Closing a Cursor frees the resources associated with the Cursor.
You cannot FETCH from a closed Cursor.
You cannot close an already closed Cursor.
Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING
Copyright © Capgemini 2015. All Rights Reserved
14


2.4 Cursor Attributes
Cursor Attributes
Cursor Attributes:
Explicit cursor attributes return information about the execution of a multi-row query.
When an “Explicit cursor” or a “cursor variable” is opened, the rows that satisfy the associated query are identified and form the result set.
Rows are fetched from the result set.
Examples: %ISOPEN, %FOUND, %NOTFOUND, %ROWCOUNT, etc.
Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING
Copyright © Capgemini 2015. All Rights Reserved
15




**Cursor Attributes: %ISOPEN**

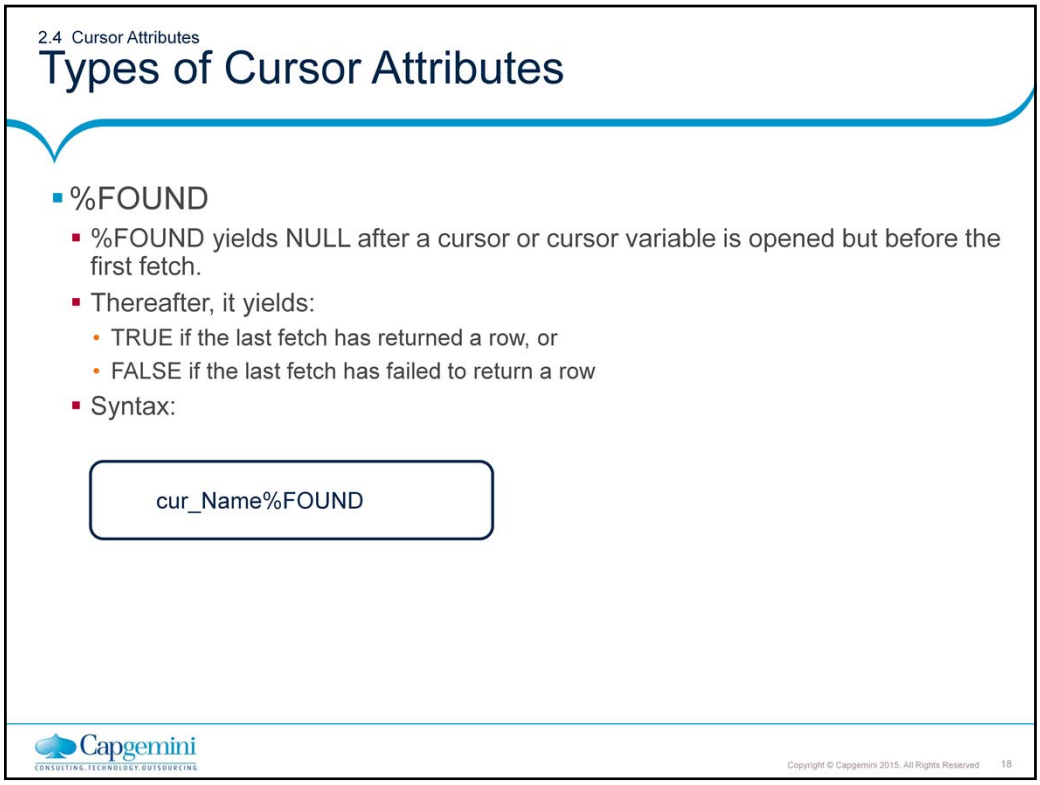
- This attribute is used to check the open/close status of a Cursor.
- If the Cursor is already open, the attribute returns TRUE.
- Oracle closes the SQL cursor automatically after executing its associated SQL statement. As a result, %ISOPEN always yields FALSE for Implicit cursor.




**Cursor Attributes: %ISOPEN (contd.)**

**Note:**

- In the example shown in the slide, C1%ISOPEN returns FALSE as the cursor is yet to be opened.
- Hence, the PL/SQL statements within the IF...END IF are not executed.




**Cursor Attributes: %FOUND:**

- Until a SQL data manipulation statement is executed, %FOUND yields NULL. Thereafter, %FOUND yields TRUE if an INSERT, UPDATE, or DELETE statement affected one or more rows, or a SELECT INTO statement returned one or more rows. Otherwise, %FOUND yields FALSE

2.4 Cursor Attributes

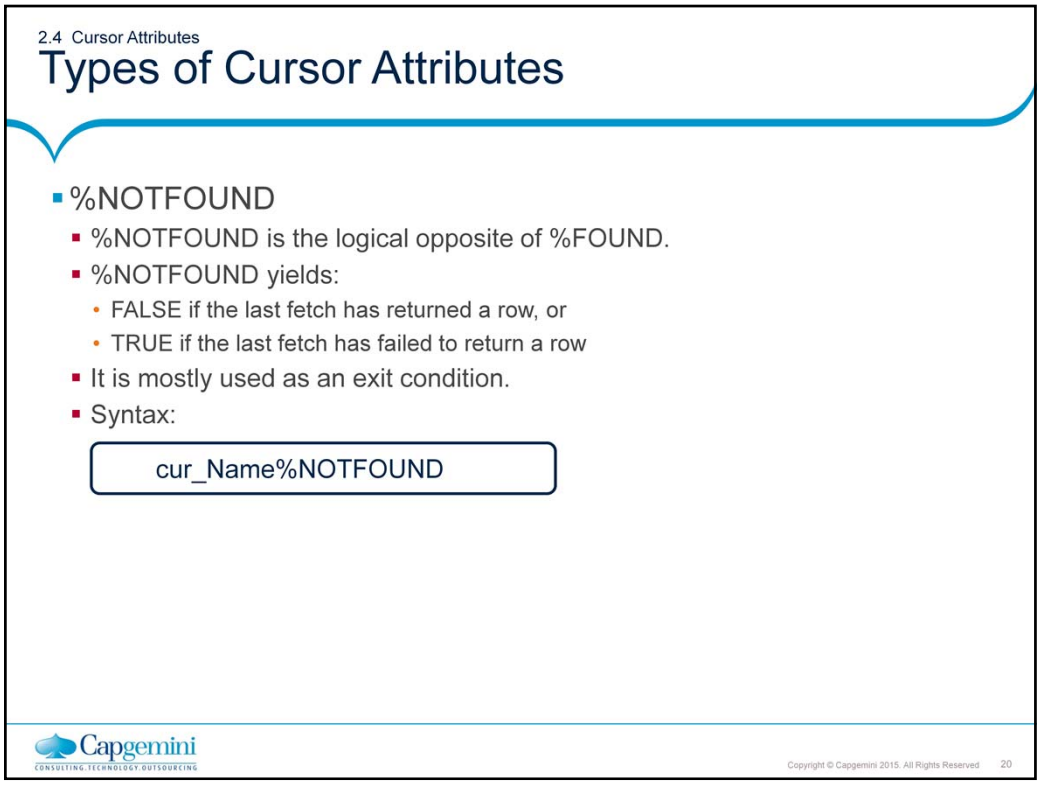
## Types of Cursor Attributes

- Example:

```
DECLARE section;
   open c1 ;
   fetch c1 into var_list ;
IF c1%FOUND THEN

      pl/sql_statements ;
END IF ;
```

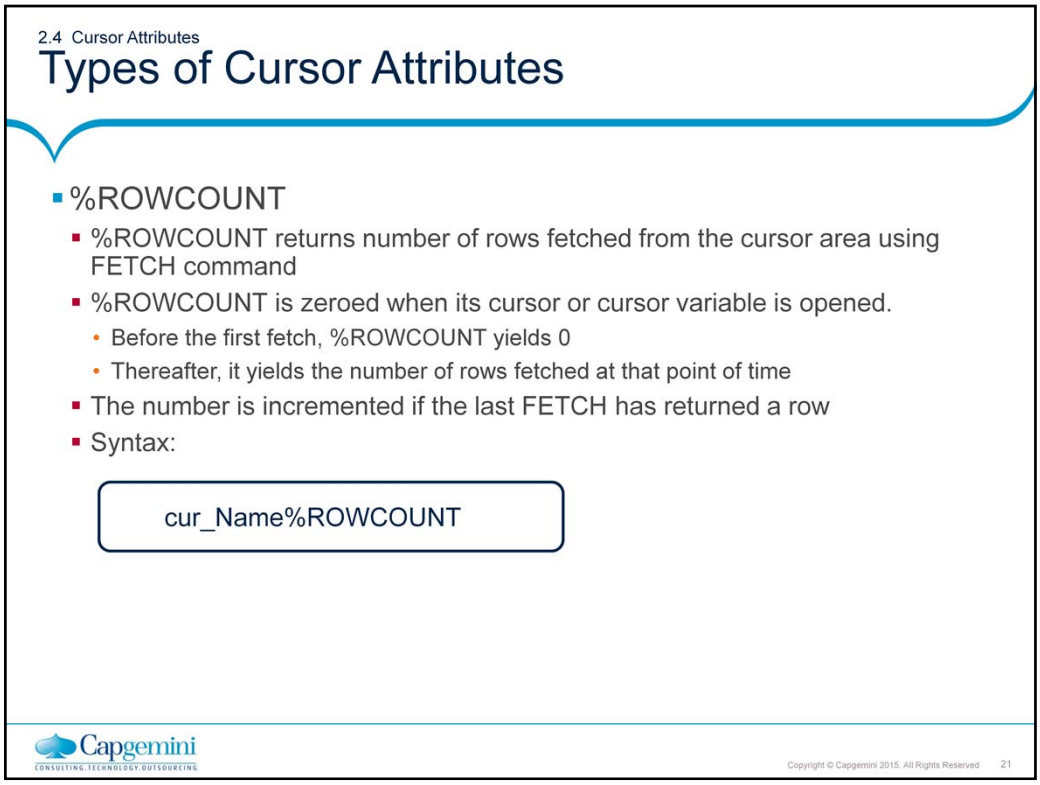
Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 19




**Cursor Attributes: %NOTFOUND:**

- %NOTFOUND is the logical opposite of %FOUND. %NOTFOUND yields TRUE if an INSERT, UPDATE, or DELETE statement affected no rows, or a SELECT INTO statement returned no rows. Otherwise, %NOTFOUND yields FALSE.




**Cursor Attributes: %ROWCOUNT**

For example: To give a 10% raise to all employees earning less than Rs. 2500.

```
DECLARE
        V_Salary   emp.sal%TYPE;
        V_Empno  emp.empno%TYPE;
        CURSOR C_Empsal  IS
        SELECT empno, sal FROM emp WHERE sal
        <2500;
BEGIN
        IF  NOT C_Empsal%ISOPEN THEN
         OPEN  C_Empsal  ;
        END  IF ;
        LOOP
        FETCH  C_Empsal  INTO V_Empno,V_Salary;
        EXIT WHEN  C_Empsal %NOTFOUND ;
                        --Exit out of block when no rows
        UPDATE emp SET sal = 1.1 * V_Salary
        WHERE empno = V_Empno;
        END LOOP ;
        CLOSE  C_Empsal ;
        COMMIT ;
END ;
```

2.4 Cursor Attributes

## Summary of Cursor Attributes

| Cursor Attribute | Return Type | Description | Syntax |
|---|---|---|---|
| %ISOPEN | Boolean | returns TRUE if its cursor or cursor variable is open. Otherwise it returns FALSE | cur_Name%ISOPEN |
| %FOUND | Boolean | returns TRUE if the last fetch has returned a row, or FALSE if the last fetch has failed to return a row | cur_Name%FOUND |
| %NOTFOUND | Boolean | returns FALSE if the last fetch has returned a row, or TRUE if the last fetch has failed to return a row | cur_Name%NOTFOUND |
| %ROWCOUNT | Nunber | Before the first fetch, %ROWCOUNT yields 0<br>Thereafter, it yields the number of rows fetched at that point of time | cur_Name%ROWCOUNT |

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 22

2.5 Processing cursors

## Cursor FETCH loops

- They are examples of simple loop statements
- The FETCH statement should be followed by the EXIT condition to avoid infinite looping
- Condition to be checked is cursor%NOTFOUND

- Examples: LOOP .. END LOOP, WHILE LOOP, etc

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 23

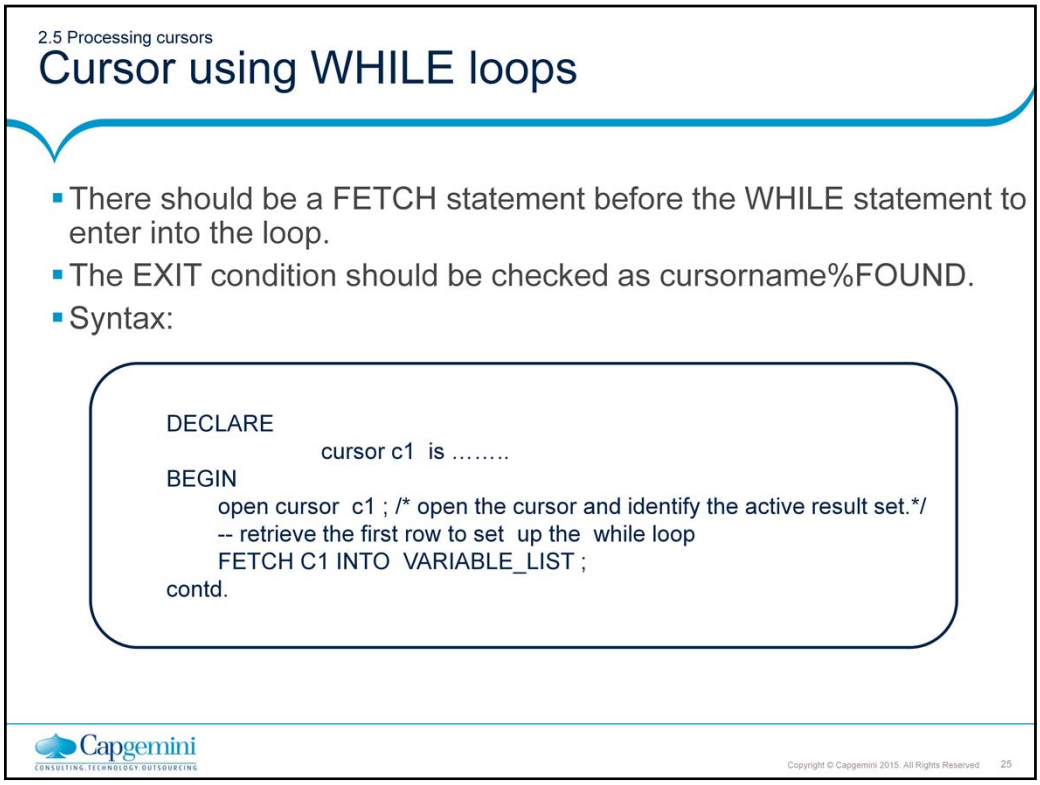
2.5 Processing cursors

## Cursor using LOOP … END LOOP:

```
DECLARE
      cursor c1 is ……..
BEGIN
     open cursor c1;  /* open the cursor and identify the active result set.*/
LOOP
     fetch c1 into  variable_list ;
     -- exit out of the loop when there are no more rows.
     /* exit is done before processing to prevent handling of null rows.*/
     EXIT WHEN C1%NOTFOUND ;
     /* Process the fetched rows using variables and  PL/SQLstatements */
END LOOP;
     -- Free resources used by the cursor
     close  c1;
     -- commit
     commit;
END;
```

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 24




**Processing Explicit Cursors: Cursor using WHILE loops:**

**Note:**

- FETCH statement appears twice, once before the loop and once after the loop processing.
- This is necessary so that the loop condition will be evaluated for each iteration of the loop.

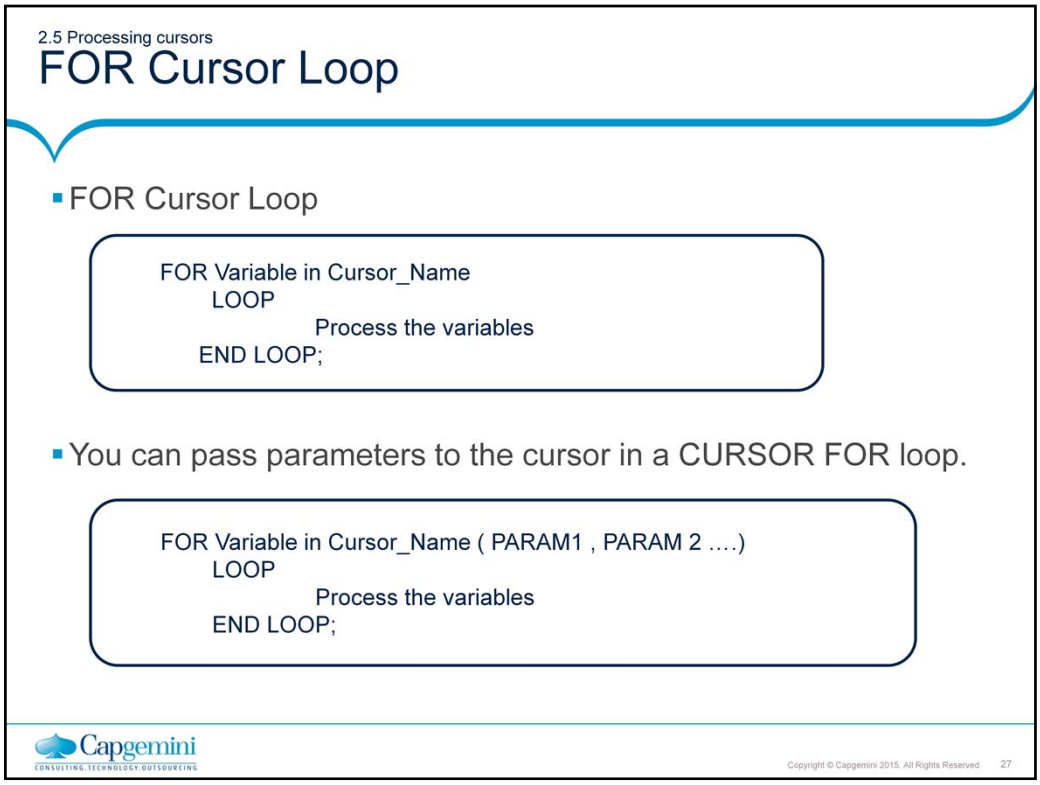
2.5 Processing cursors

## Cursor using WHILE loops…contd

```
/*Continue  looping ,  processing & fetching till last row is
retrieved.*/
WHILE C1%FOUND
LOOP
      fetch c1 into variable_list;
END LOOP;
   CLOSE C1; -- Free resources used by the cursor.
   commit;   -- commit
END;
```

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 26




**Processing Explicit Cursors: FOR CURSOR Loop:**

- For all other loops we had to go through all the steps of OPEN, FETCH, AND CLOSE statements for a cursor.
- PL/SQL provides a shortcut via a CURSOR FOR Loop. The CURSOR FOR Loop implicitly handles the cursor processing.

```
DECLARE
CURSOR  C1  IS ……..
BEGIN
-- An implict Open  of the  cursor C1 is done here.
-- Record variable should not be declared in declaration
section
FOR  Record_Variable   IN  C1 LOOP
-- An implicit Fetch is done here
-- Process the fetched rows using variables  and  PL/SQL
statements
-- Before the loop is continued an implicit C1%NOTFOUND
test is done by PL/SQL
END LOOP;
-- An automatic close  C1 is issued after termination of the
loop
-- Commit
COMMIT ;
END;
/
```

- In this snippet, the record variable is implicitly declared by PL/SQL and is of the type C1%ROWTYPE and the scope of Record_Variable is only for the cursor FOR LOOP.

2.5 Processing cursors

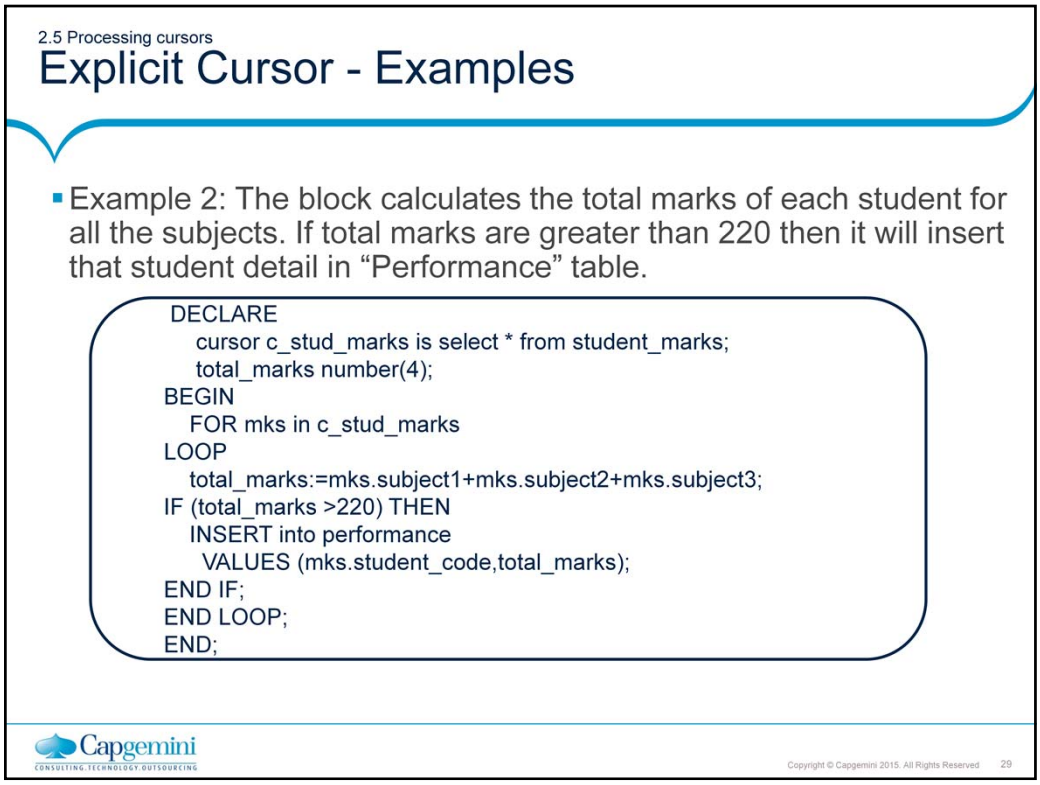
## Explicit Cursor - Examples

- Example 1: To add 10 marks in subject3 for all students

```
DECLARE
      v_sno      student_marks.student_code%type;
      cursor c_stud_marks is
     select student_code from student_marks;
BEGIN
      OPEN c_stud_marks;
      FETCH c_stud_marks into v_sno;
      WHILE c_stud_marks%found
        LOOP
          UPDATE student_marks SET subject3 =subject3+10
         WHERE student_code = v_sno ;
   FETCH c_stud_marks into v_empno;
   END LOOP ;
   CLOSE c_stud_marks;
END;
```

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 28




In the above example “Performance” is a user defined table.

## Summary

- In this lesson, you have learnt:
  - Cursor is a “handle” or “name” for a private SQL area
  - Implicit cursors are declared for queries that return only one row
  - Explicit cursors are declared for queries that return more than one row


Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 30

**Answers for Review Questions:**

**Question 1:**
Answer: False

**Question 2:**
Answer: False

**Question 3:**
Answer: CURSOR FOR

## Review – Questions

- Question 1 : %COUNT returns number of rows fetched from the cursor area by using FETCH command.
  - True / False
- Question 2: Implicit SQL cursor is opened or closed by the program.
  - True / False
- Question 3: PL/SQL provides a shortcut via a ___ Loop, which implicitly handles the cursor processing.

# Oracle Basics (PL/SQL)

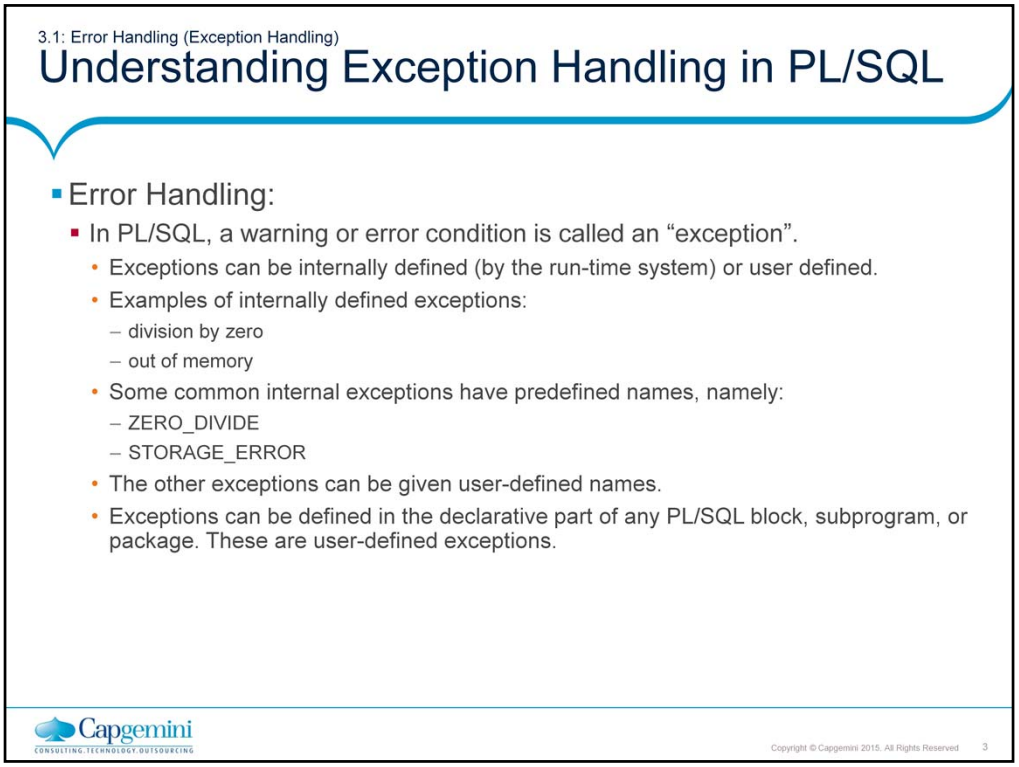
Lesson 03 Exception Handling

# Lesson Objectives

- To understand the following topics:
  - Error Handling
  - Declaring Exceptions
    - Predefined Exceptions
    - User Defined Exceptions
      - Raising Exceptions
  - OTHERS exception handler

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 2

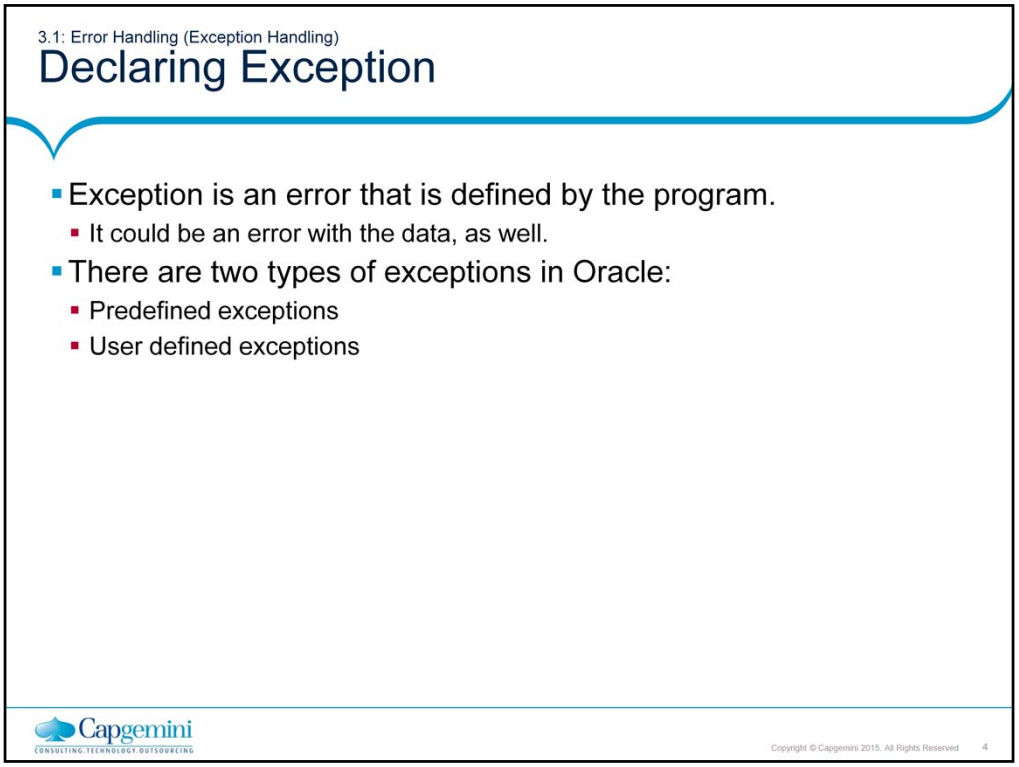



**Error Handling:**

- A good programming language should provide capabilities of handling errors and recovering from them if possible.
- PL/SQL implements Error Handling via “exceptions” and “exception handlers”.

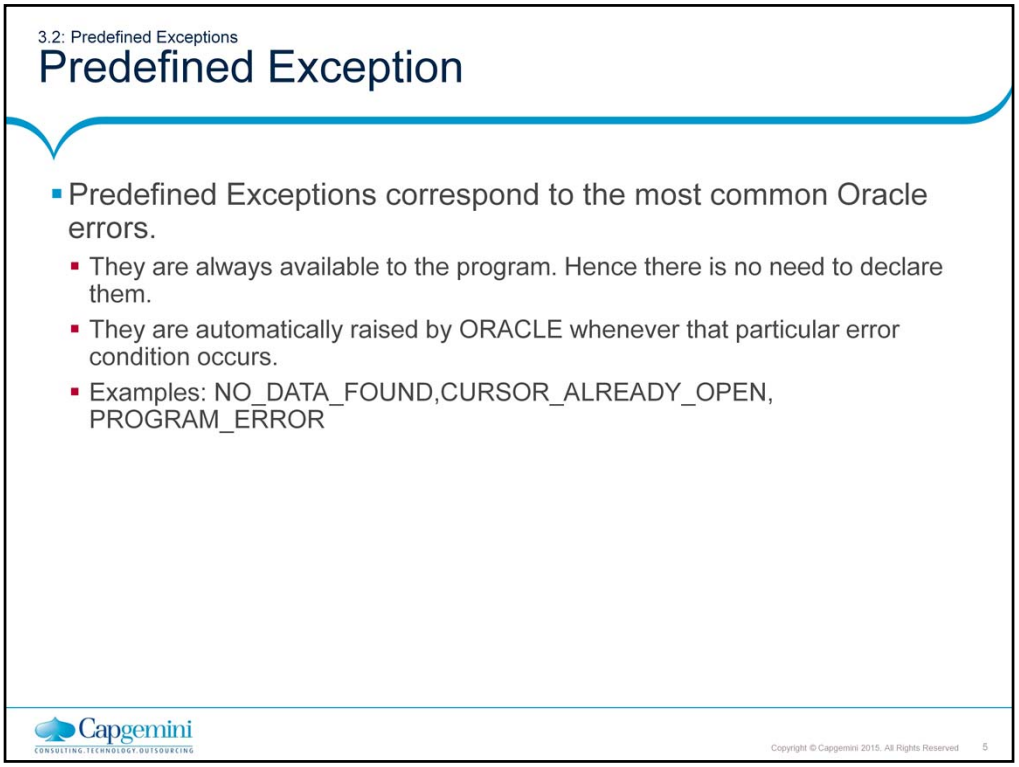
**Types of Errors in PL/SQL**

- **Compile Time errors:** They are reported by the PL/SQL compiler, and you have to correct them before recompiling.
- **Run Time errors:** They are reported by the run-time engine. They are handled programmatically by raising an exception, and catching it in the Exception section.



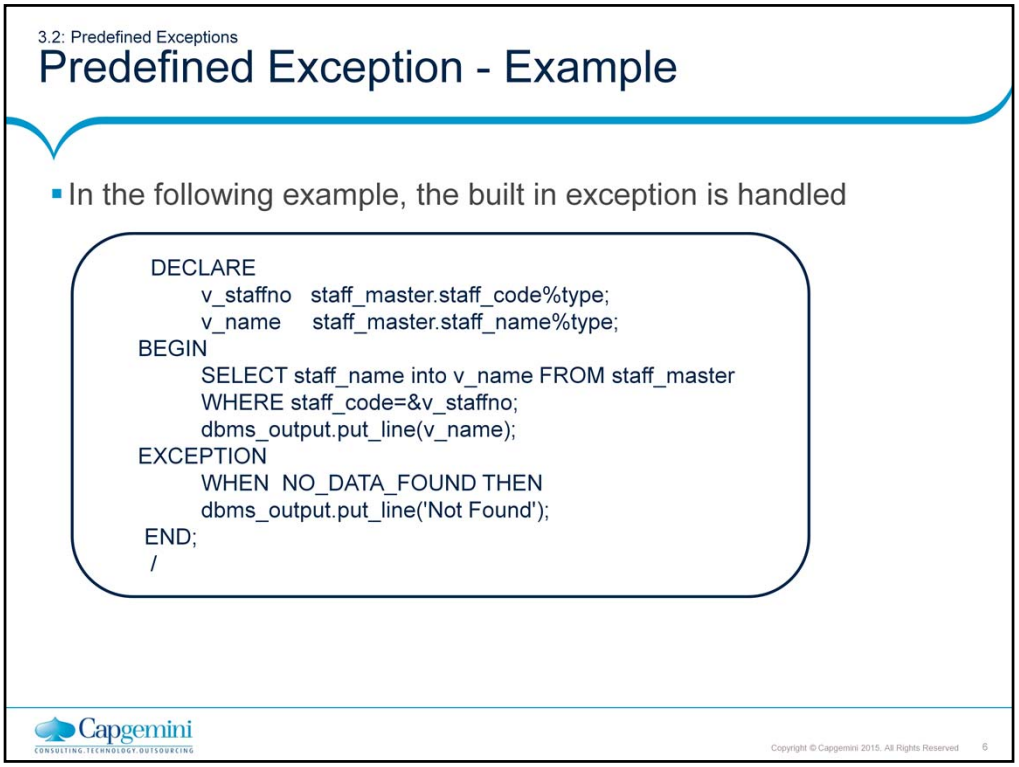

**Declaring Exceptions:**

- Exceptions are declared in the Declaration section, raised in the Executable section, and handled in the Exception section.

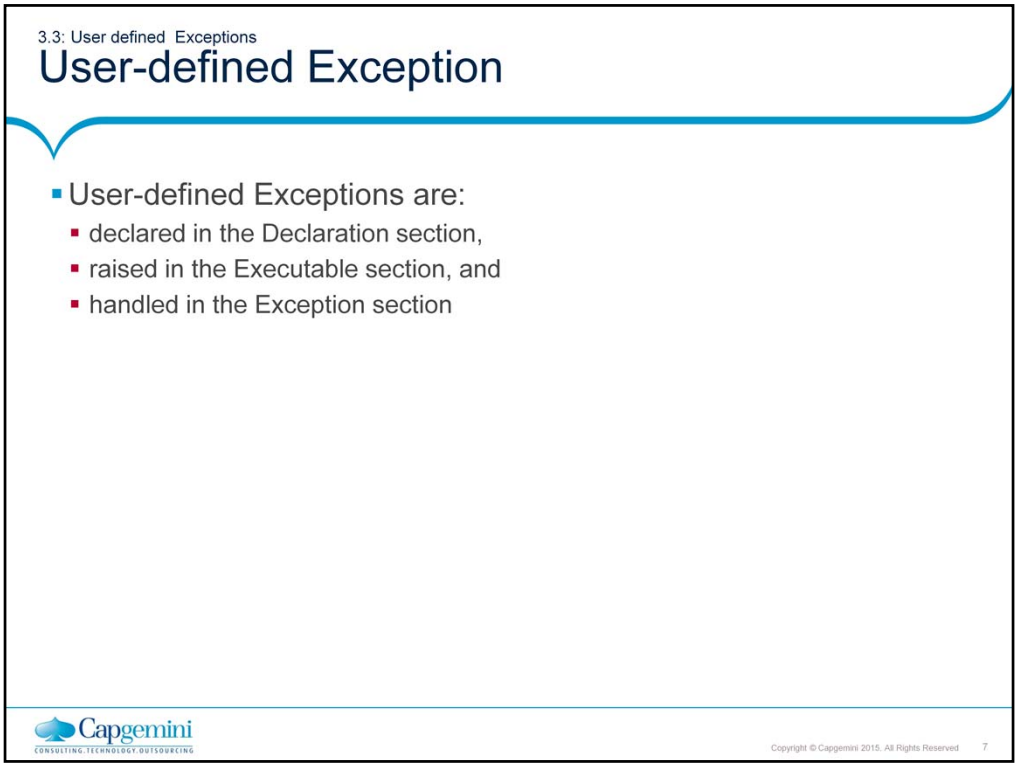



**Predefined Exceptions:**

- An internal exception is raised implicitly whenever your PL/SQL program violates an Oracle rule or exceeds a system-dependent limit. Every Oracle error has a number, but exceptions must be handled by name. So, PL/SQL predefines some common Oracle errors as exceptions. For example, PL/SQL raises the predefined exception NO_DATA_FOUND if a SELECT INTO statement returns no rows.
- Given below are some Predefined Exceptions:
  - NO_DATA_FOUND
    - This exception is raised when SELECT INTO …. statement does not return any rows.
  - TOO_MANY_ROWS
    - This exception is raised when SELECT INTO …. statement returns more than one row.
  - INVALID_CURSOR
    - This exception is raised when an illegal cursor operation is performed such as closing an already closed cursor.
  - VALUE_ERROR
    - This exception is raised when an arithmetic, conversion, truncation, or constraint error occurs in a procedural statement.
  - DUP_VAL_ON_INDEX
    - This exception is raised when the UNIQUE CONSTRAINT is violated.




**Predefined Exceptions:**
In the example shown on the slide, the NO_DATA_FOUND built in exception is handled. It is automatically raised if the SELECT statement does not fetch any value and populate the variable.




**User-Defined Exceptions:**

- These exception are entirely user defined based on the application. The programmer is responsible for declaring, raising and handling them.

3.3: User defined Exceptions

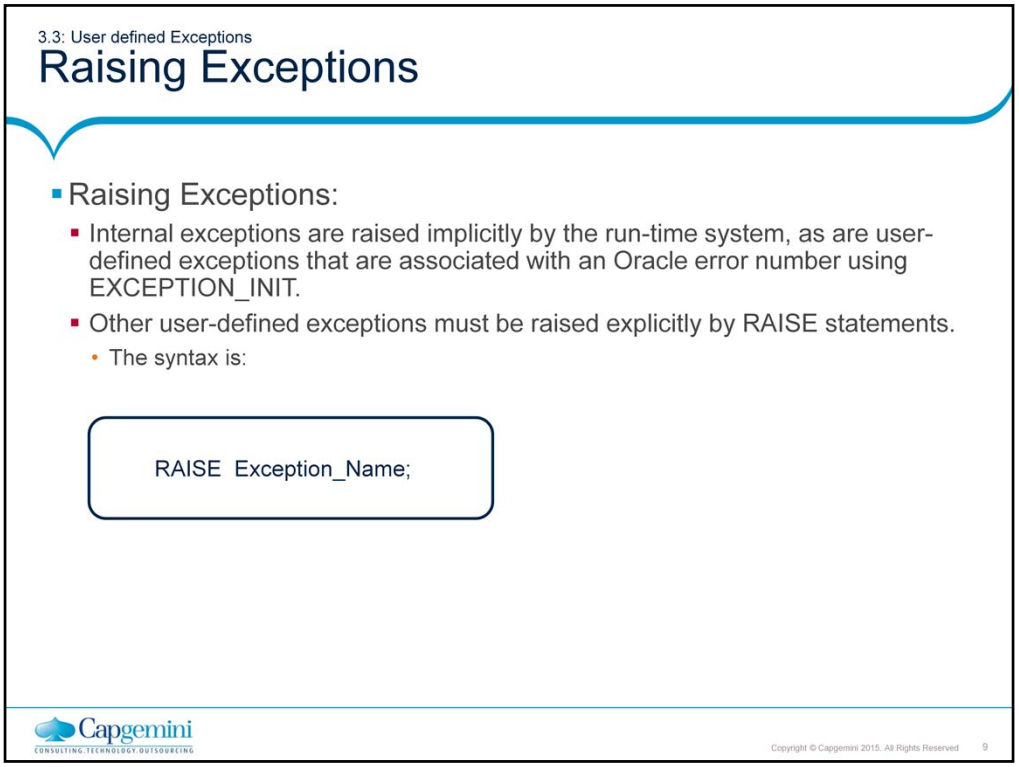
## User-defined Exception - Example

- Here is an example of User Defined Exception:

```
DECLARE
      E_Balance_Not_Sufficient EXCEPTION;
      E_Comm_Too_Large EXCEPTION;
      …
BEGIN
      NULL;
END;
```

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 8




**Raising Exceptions:**
When the error associated with an exception occurs, the exception is raised. This is done through the RAISE command.

3.3: User defined Exceptions

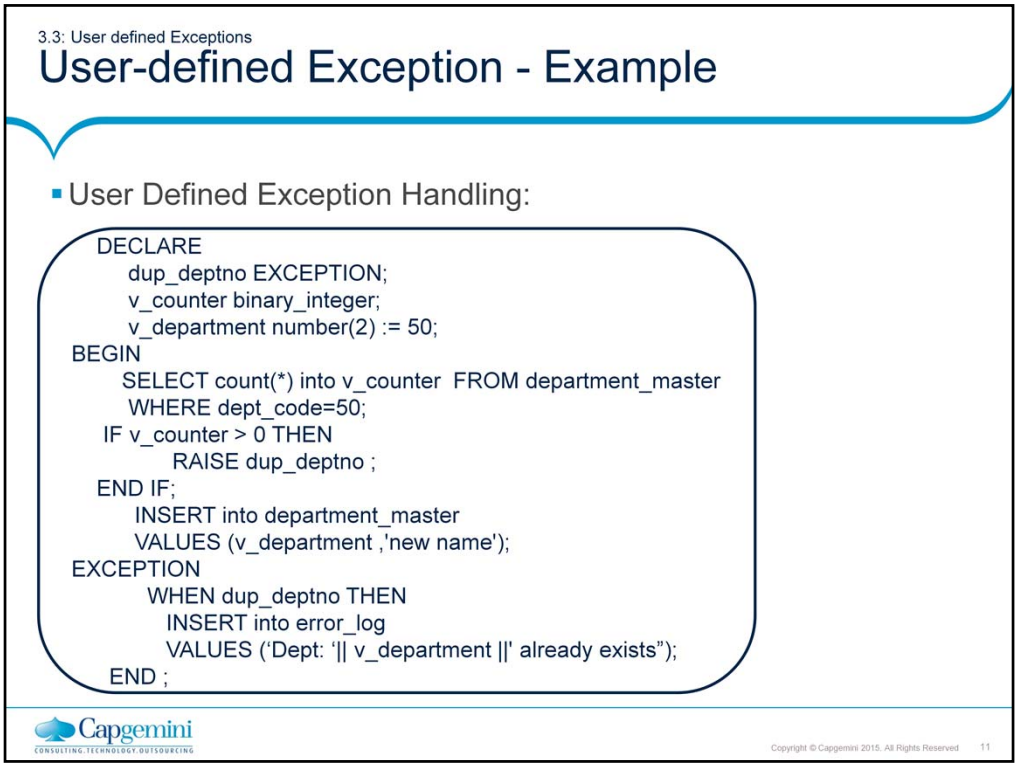
## Raising Exceptions - Example

- An exception is defined and raised as shown below:

```
DECLARE
      ...
      retired_emp EXCEPTION ;
BEGIN
      pl/sql_statements ;
      if error_condition then
                  RAISE retired_emp ;
                              pl/sql_statements ;
EXCEPTION
      WHEN retired_emp THEN
      pl/sql_statements ;
END ;
```

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 10




The example on the slide demonstrates user-defined exceptions. It checks for department no value to be inserted in the table. If the value is duplicated it will raise an exception.

3.4: OTHERS Exception Handler

## OTHERS Exception Handler

- OTHERS Exception Handler:
  - The optional OTHERS exception handler, which is always the last handler in a block or subprogram, acts as the handler for all exceptions that are not specifically named in the Exception section.
  - A block or subprogram can have only one OTHERS handler.

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 12

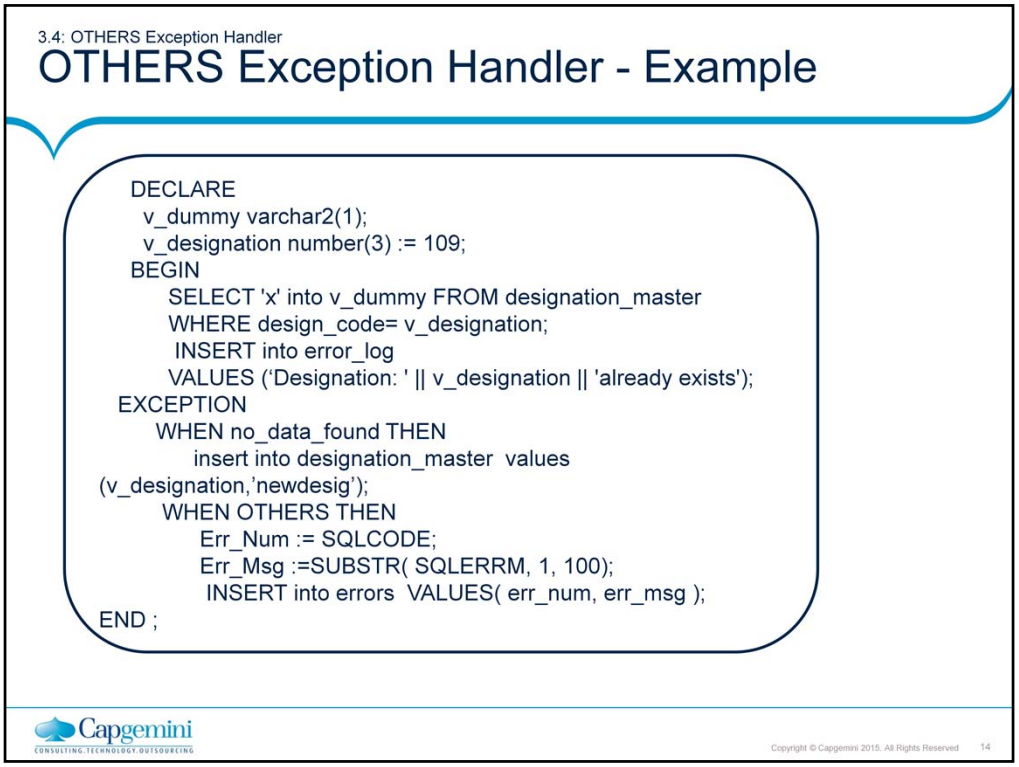
3.4: OTHERS Exception Handler

## OTHERS Exception Handler (contd..)

- To handle a specific case within the OTHERS handler, predefined functions SQLCODE and SQLERRM are used.
  - SQLCODE returns the current error code. And SQLERRM returns the current error message text.
  - The values of SQLCODE and SQLERRM should be assigned to local variables before using it within a SQL statement.

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 13

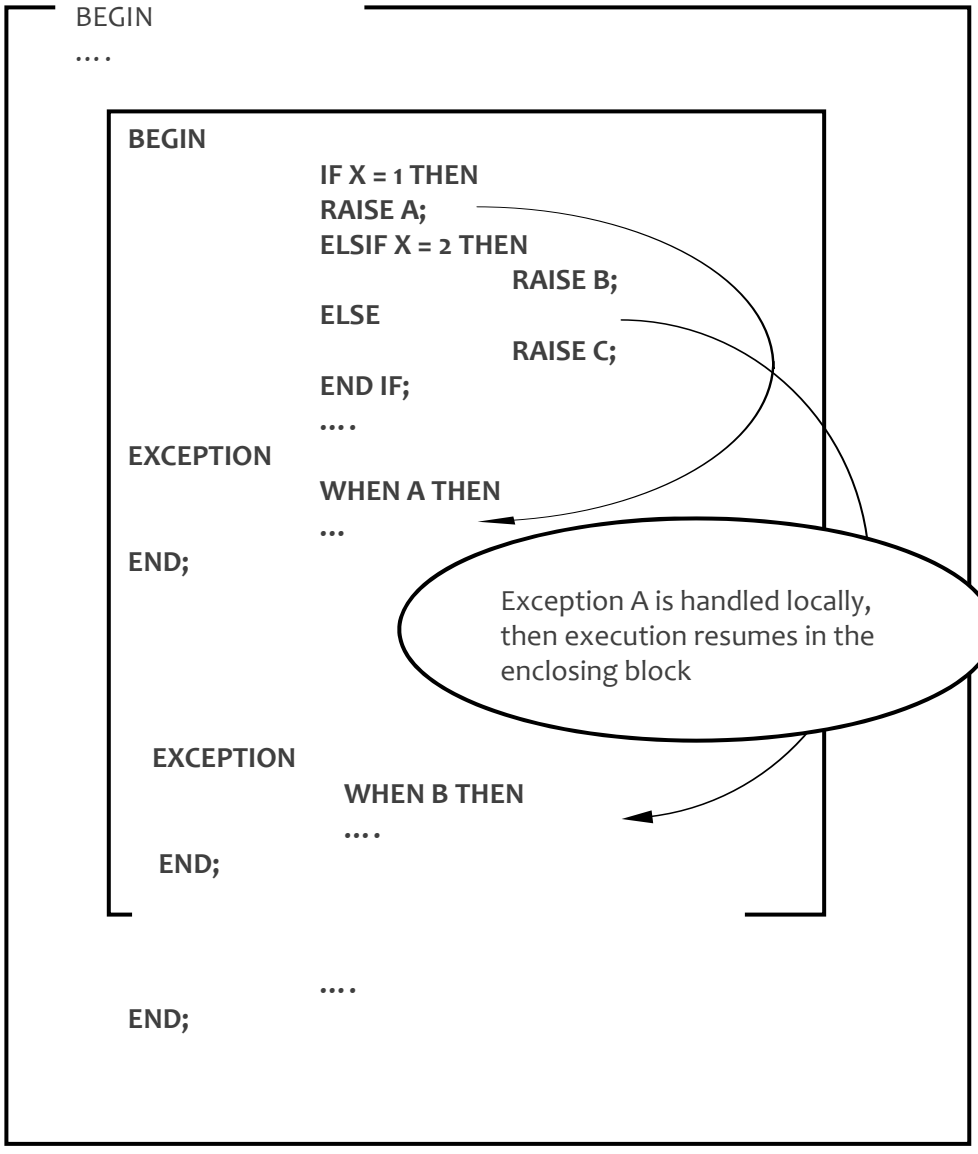



The example on the slide uses OTHERS Exception handler. If the exception that is raised by the code is not NO_DATA_FOUND, then it will go to the OTHERS exception handler since it will notice that there is no appropriate exception handler defined.

Also observe that the values of SQLCODE and SQLERRM are assigned to variables defined in the block.

**Propagation of Exceptions:**

- When an exception is raised, if PL/SQL cannot find a handler for it in the current block or subprogram, then the exception propagates. That is, the exception reproduces itself in successive enclosing blocks until a handler is found or there are no more blocks to search.



**Masking Location of an Error:**

- Since the same Exception section is examined for the entire block, it can be difficult to determine, which SQL statement caused the error.

```
SELECT
SELECT
SELECT
EXCEPTION
WHEN  NO_DATA_FOUND THEN
--You Don't Know  which caused the NO_DATA_FOUND
END ;
```

```
DECLARE
V_Counter  NUMBER:= 1;
BEGIN
SELECT ……
V_Counter := 2;
SELECT  ……
V_Counter :=3;
SELECT  …
WHEN NO_DATA_FOUND THEN
 -- Check values of V_Counter to  find out which SELECT
statement
 -- caused the exception NO_DATA_FOUND
END ;
```

```
BEGIN
-- PL/SQL Statements
BEGIN
SELECT ….
EXCEPTION
WHEN NO_DATA_FOUND THEN
      --
END;
BEGIN
SELECT ….
EXCEPTION
WHEN NO_DATA_FOUND THEN
      --
END;
BEGIN
SELECT ….
EXCEPTION
WHEN NO_DATA_FOUND THEN
      --
END;
END ;
```

**Masking Location of an Error (contd.):**

```
BEGIN
--------
./* PL/SQL statements */
BEGIN
SELECT ……..
WHEN  NO_DATA_FOUND  THEN
-- Process  the error for NO_DATA_FOUND
END;

/* Some more PL/SQL statements
This will execute irrespective of when
NO_DATA_FOUND */
 END;
```

## Summary

- In this lesson, you have learnt about:
  - Exception Handling
    - Predefined Exceptions
    - User-defined Exceptions
  - OTHERS exception handler


Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 18

## Review – Questions

- Question 1: ___ returns the current error message text.
- Question 2: ___ returns the current error code.

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 19

# Oracle Basics (PL/SQL)

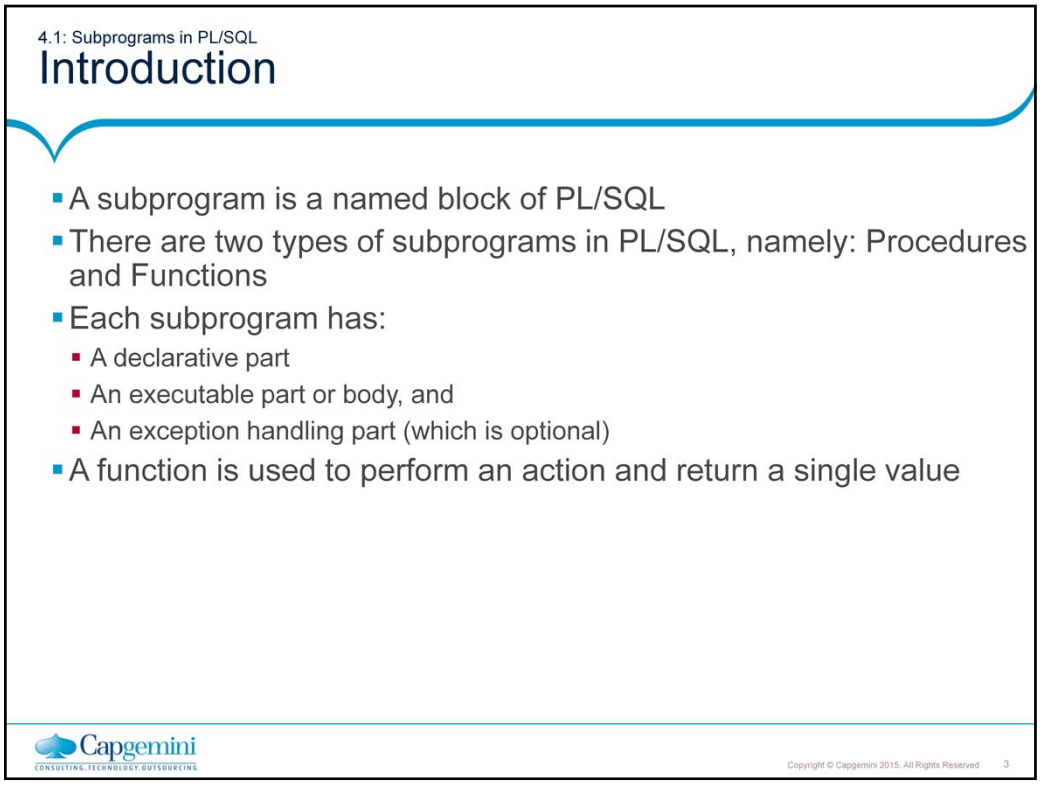
Lesson 04 Procedures and Functions

## Lesson Objectives

- To understand the following topics:
  - Subprograms in PL/SQL
  - Anonymous blocks versus Stored Subprograms
  - Procedure
    - Subprogram Parameter modes
  - Functions

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 2




**Subprograms in PL/SQL:**

- The subprograms are compiled and stored in the Oracle database as "stored programs", and can be invoked whenever required. As the subprograms are stored in a compiled form, when called they only need to be executed. Hence this arrangement saves time needed for compilation.
- When a client executes a procedure or function, the processing is done in the server. This reduces the network traffic.
- Subprograms provide the following advantages:
  - They allow you to write a PL/SQL program that meets our need.
  - They allow you to break the program into manageable modules.
  - They provide reusability and maintainability for the code.

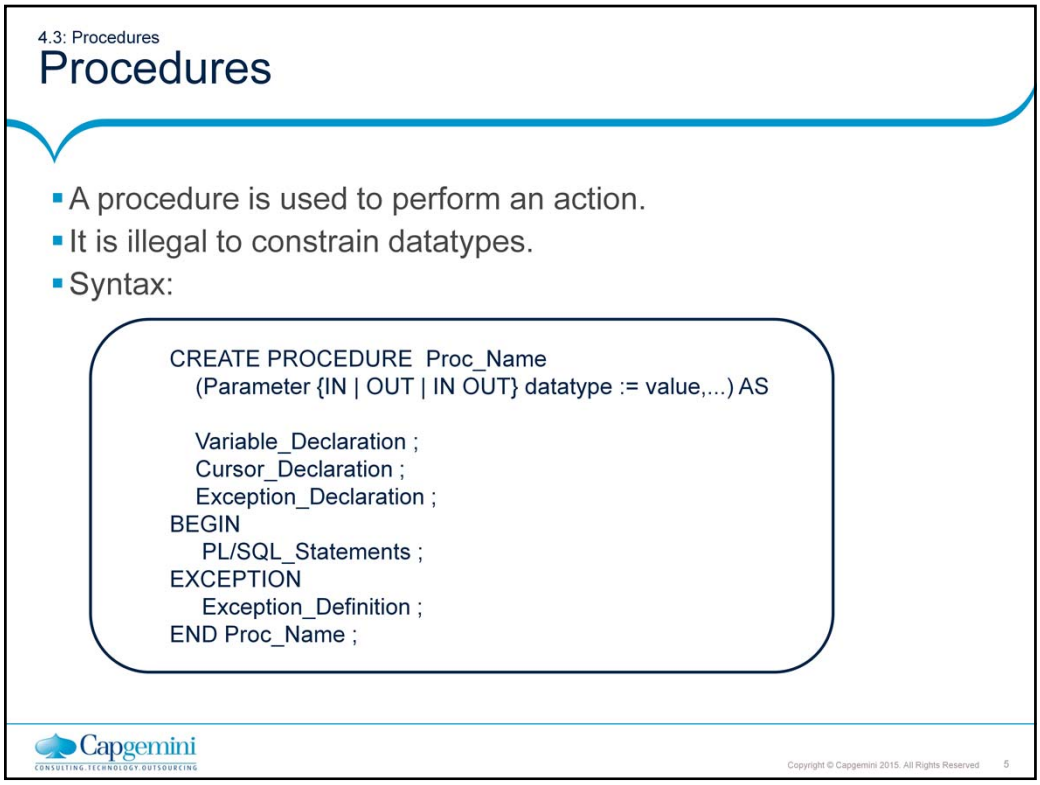
4.2: Anonymous Blocks & Stored Subprograms

# Anonymous Blocks & Stored Subprograms Comparison

| Anonymous Blocks | Stored Subprograms/Named Blocks |
|---|---|
| 1. Anonymous Blocks do not have names. | 1. Stored subprograms are named PL/SQL blocks. |
| 2. They are interactively executed. The block needs to be compiled every time it is run. | 2. They are compiled at the time of creation and stored in the database itself. Source code is also stored in the database. |
| 3. Only the user who created the block can use the block. | 3. Necessary privileges are required to execute the block. |

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

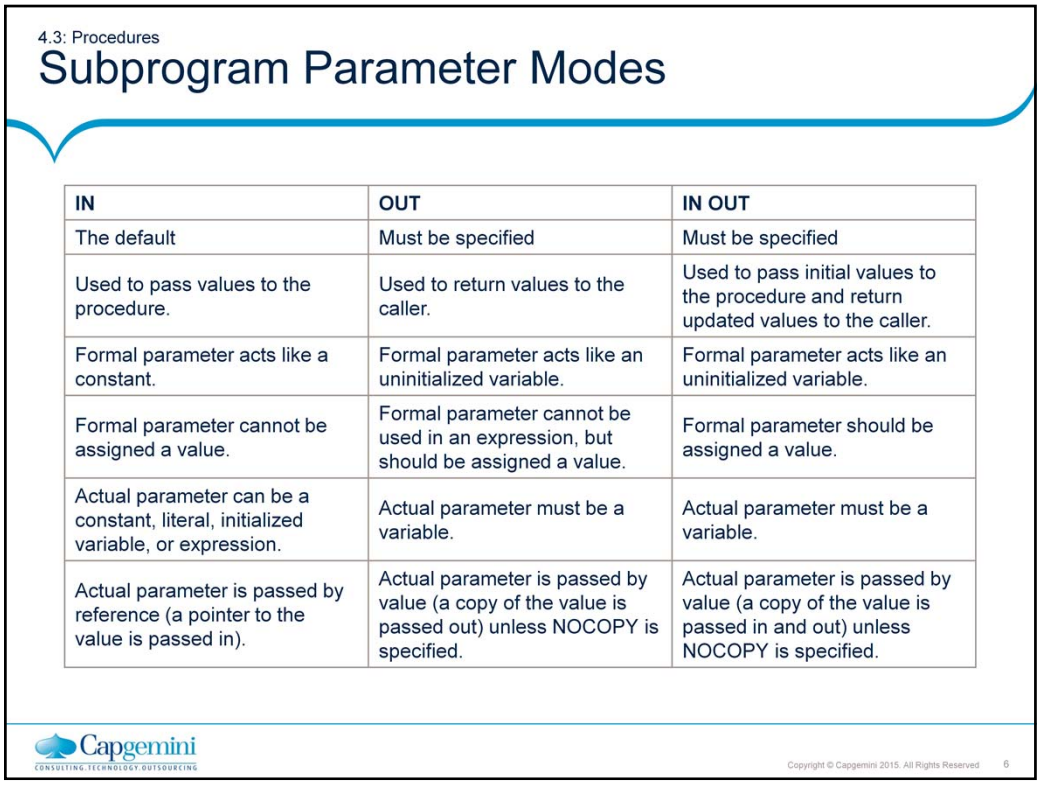
 4




**Procedures:**

- A procedure is a subprogram used to perform a specific action.
- A procedure contains two parts:
  - the specification, and
  - the body
- The procedure specification begins with CREATE and ends with procedure name or parameters list. Procedures that do not take parameters are written without a parenthesis.
- The procedure body starts after the keyword IS or AS and ends with keyword END.

contd.




**Subprogram Parameter Modes:**

- You use “parameter modes” to define the behavior of “formal parameters”. The three parameter modes are IN (the default), OUT, and INOUT. The characteristics of the three modes are shown in the slide.
- Any parameter mode can be used with any subprogram.
- Avoid using the OUT and INOUT modes with functions.
- To have a function return multiple values is a poor programming practice. Besides functions should be free from side effects, which change the values of variables that are not local to the subprogram.
- Example1:

```
CREATE PROCEDURE split_name
(
  phrase IN VARCHAR2, first OUT VARCHAR2, last
OUT VARCHAR2
)
IS
  first := SUBSTR(phrase, 1, INSTR(phrase, ' ')-1);
  last := SUBSTR(phrase, INSTR(phrase, ' ')+1);
  IF first = 'John' THEN
    DBMS_OUTPUT.PUT_LINE('That is a common first
name.');
  END IF;
END;
```

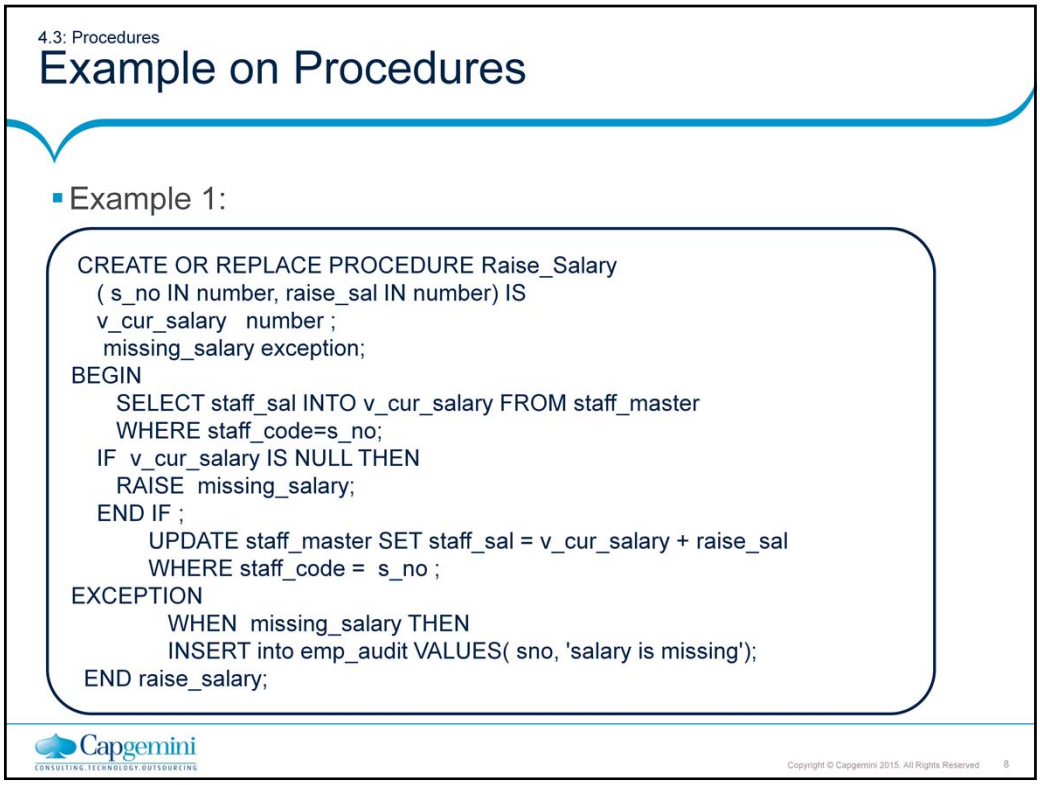
**Subprogram Parameter Modes (contd.):**
**Examples:**

Example 2:

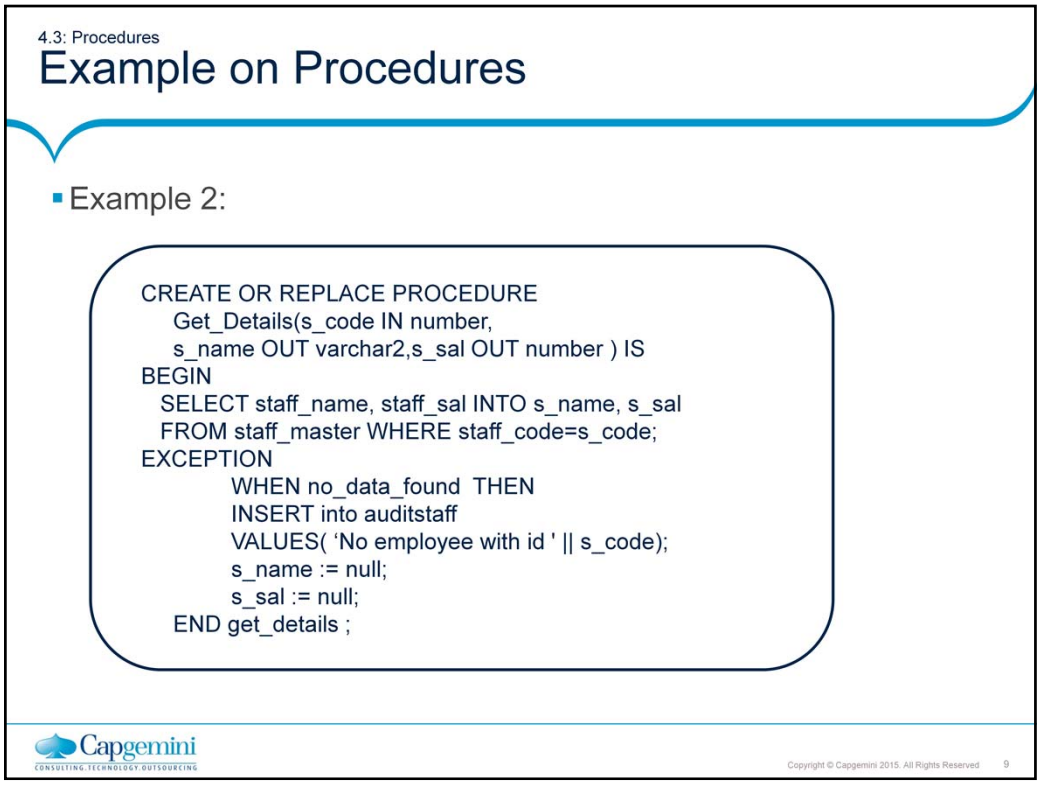
```
SQL > SET SERVEROUTPUT ON
SQL > CREATE OR REPLACE PROCEDURE PROC1 AS
    2   BEGIN
    3       DBMS_OUTPUT.PUT_LINE('Hello from procedure …');
    4   END;
    5   /
Procedure created.
SQL > EXECUTE PROC1
Hello from procedure …
PL/SQL procedure successfully created.
```

Example 3:

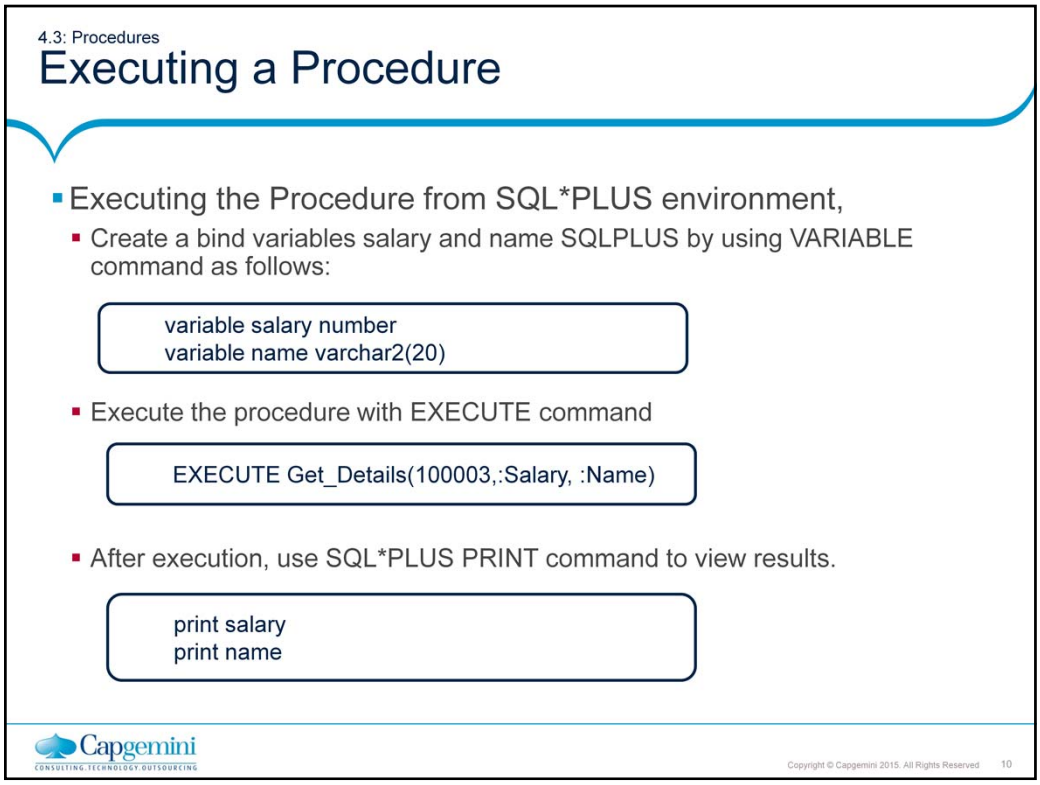
```
SQL > CREATE OR REPLACE       PROCEDURE PROC2
    2       (N1 IN NUMBER, N2 IN NUMBER, TOT OUT NUMBER) IS
    3   BEGIN
    4       TOT := N1 + N2;
    5   END;
    6   /

Procedure created.

SQL > VARIABLE T NUMBER
SQL > EXEC PROC2(33, 66, :T)

PL/SQL procedure successfully completed.

SQL > PRINT T

           T
-- - - - - - - - - -
           99
```




The procedure example on the slide modifies the salary of staff member. It also handles exceptions appropriately. In addition to the above shown exception you can also handle “NO_DATA_FOUND” exception. The procedure accepts two parameters which is the staff_code and amount that has to be given as raise to the staff member.




The procedure on the slide accept three parameters, one is IN mode and other two are OUT mode. The procedure retrieves the name and salary of the staff member based on the staff_code passed to the procedure. The S_NAME and S_SAL are the OUT parameters that will return the values to the calling environment

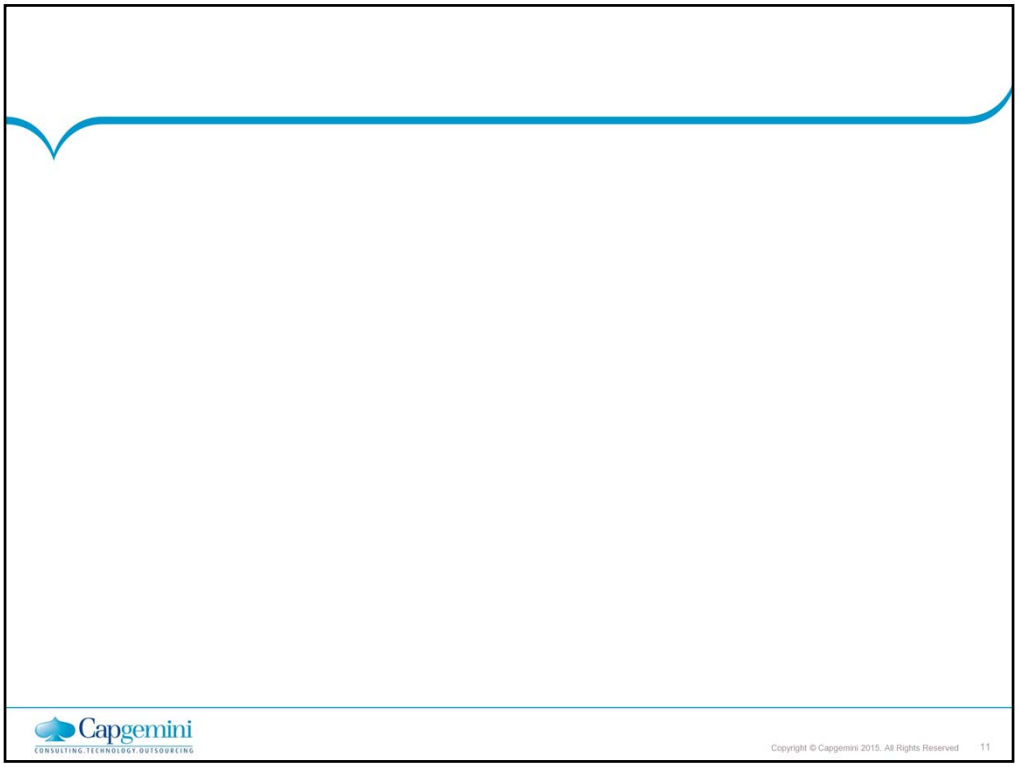



Procedures can be executed through command line as shown on the slide or can be called from other procedures/functions/Anonymous PL/SQL blocks.

On the slide the first snippet declares two variables viz. salary and name. The second snippet calls the procedure and passes the actual parameters. The first is a literal string and the next two parameters are empty variables which will be assigned with values within the procedure.

Calling the procedure from an anonymous PL/SQL block

```
DECLARE
      s_no  number(10):=&sno;
      sname varchar2(10);
      sal  number(10,2);
BEGIN
      Get_Details(s_no,sname,sal);
      dbms_output.put_line('Name:'||sname||'Salary'||sal);
END;
```




A bind variable is a variable that you declare in a host environment and then use to pass runtime values.
These values can be character or numeric. You can pass these values either in or out of one or more PL/SQL programs, such as packages, procedures, or functions.
To declare a bind variable in the SQL*Plus environment, you use the command VARIABLE.
For example,
VARIABLE salary NUMBER

Upon declaration, the **bind variables** now become **host** to that environment, and you can now use these variables within your PL/SQL programs, such as packages, procedures, or functions.
To reference host variables, you must add a prefix to the reference with a colon (:) to distinguish the host variables from declared PL/SQL variables.
example
EXECUTE Get_Details(100003,:salary, :name)

**Parameter default values:**

- Like variable declarations, the formal parameters to a procedure or function can have default values.
- If a parameter has default values, it does not have to be passed from the calling environment.
  - If it is passed, actual parameter will be used instead of default.
- Only IN parameters can have default values.

```
PROCEDURE Create_Dept( New_Deptno IN NUMBER,
   New_Dname IN  VARCHAR2  DEFAULT 'TEMP') IS
BEGIN
  INSERT INTO  department_master
        VALUES ( New_Deptno, New_Dname, New_Loc) ;
END ;
```

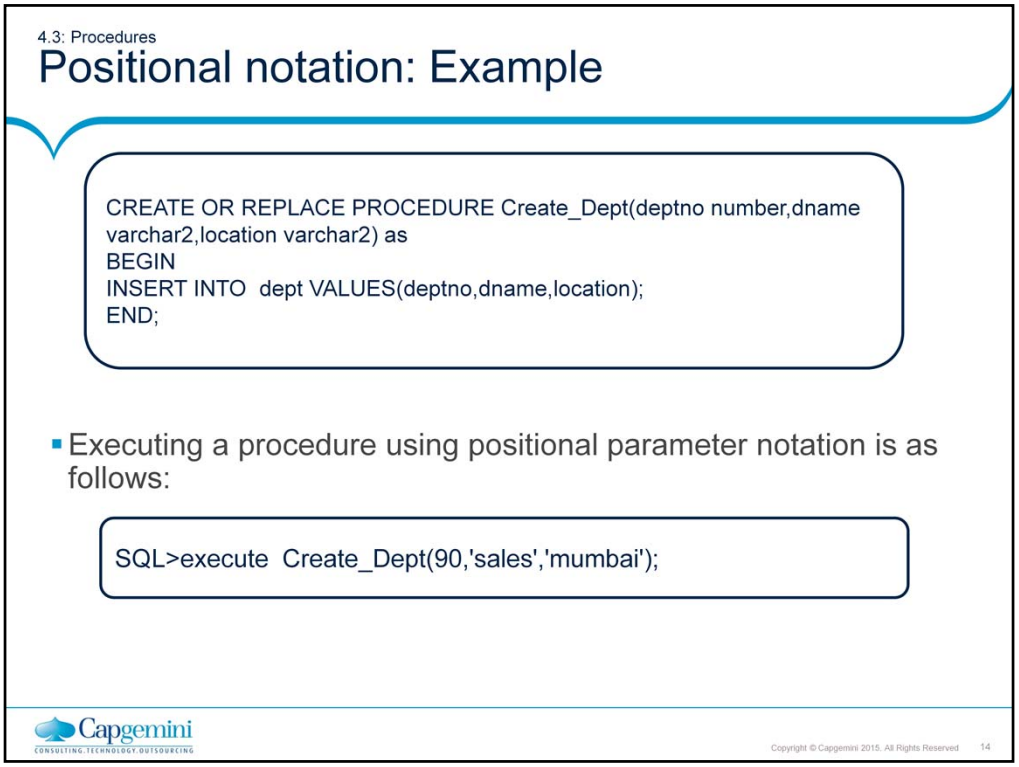
```
BEGIN
Create_Dept( 50);
--  Actual  call will be  Create_Dept ( 50, 'TEMP',
'TEMP')

Create_Dept ( 50, 'FINANCE');
-- Actual call will be  Create_Dept ( 50, 'FINANCE'
,'TEMP')

Create_Dept( 50, 'FINANCE', 'BOMBAY') ;
-- Actual  call will be  Create_Dept(50, 'FINANCE',
'BOMBAY' )

END;
```

**Procedures (contd.):**

**Using Positional, Named, or Mixed Notation for Subprogram Parameters:**

- When calling a subprogram, you can write the actual parameters by using either Positional notation, Named notation, or Mixed notation.
  - **Positional notation:** You specify the same parameters in the same order as they are declared in the procedure. This notation is compact, but if you specify the parameters (especially literals) in the wrong order, the bug can be hard to detect. You must change your code if the procedure's parameter list changes.
  - **Named notation:** You specify the name of each parameter along with its value. An arrow (=>) serves as the "association operator". The order of the parameters is not significant.
  - **Mixed notation:** You specify the first parameters with "Positional notation", and then switch to "Named notation" for the last parameters. You can use this notation to call procedures that have some "required parameters", followed by some "optional parameters".
- We have already seen a few examples of calling procedures with Positional notation.

```
Create_Dept (New_Deptno=> 50, New_Dname=>'FINANCE');
```

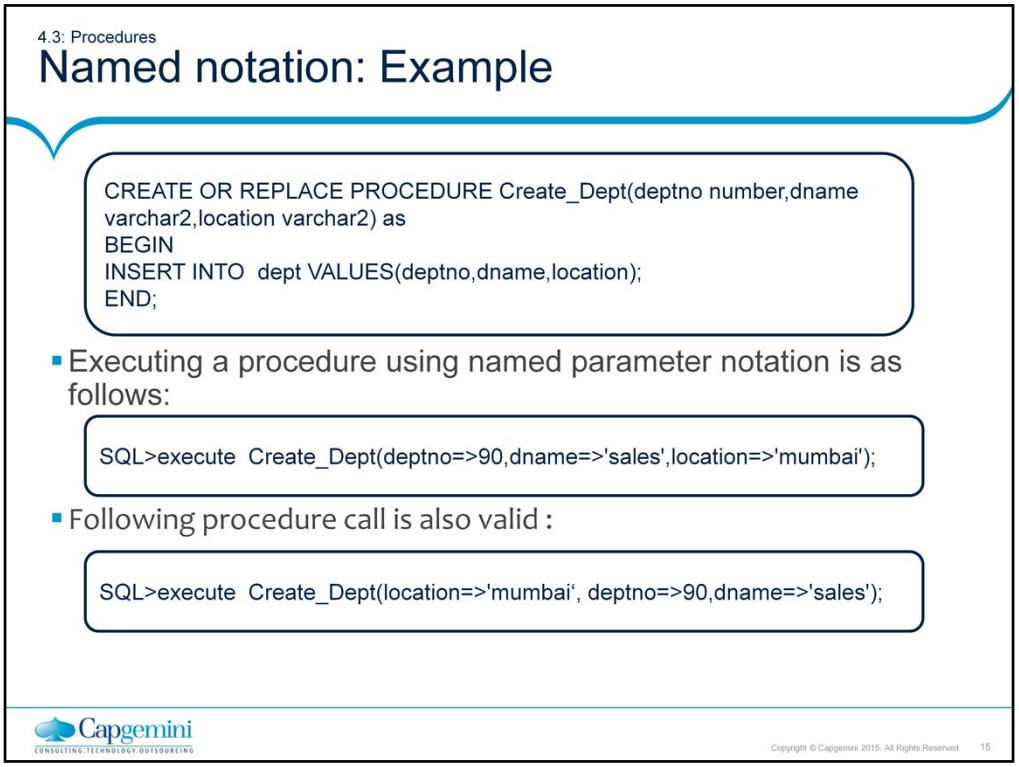



**Positional Notation :**

You specify the parameters in the same order as they are declared in the procedure.

This notation is compact, but if you specify the parameters (especially literals) in the wrong order, the bug can be hard to detect.

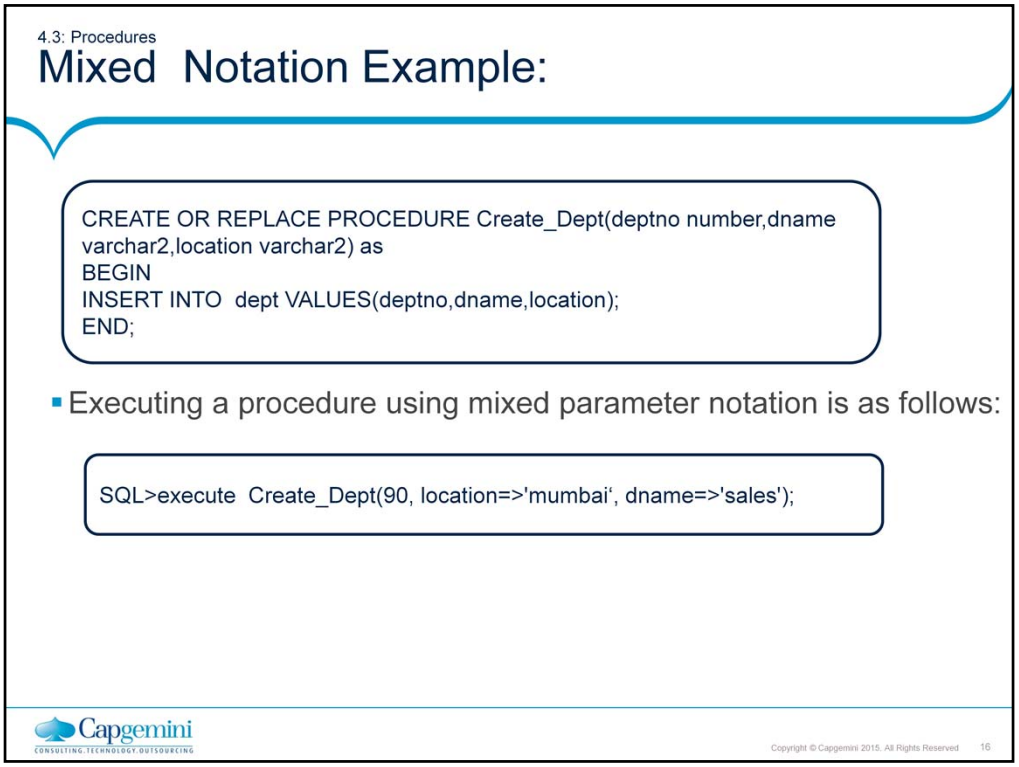
You must change your code if the procedure's parameter list changes




**Named notation:**

You specify the name of each parameter along with its value. An arrow (=>) serves as the “association operator”. The order of the parameters is not significant.

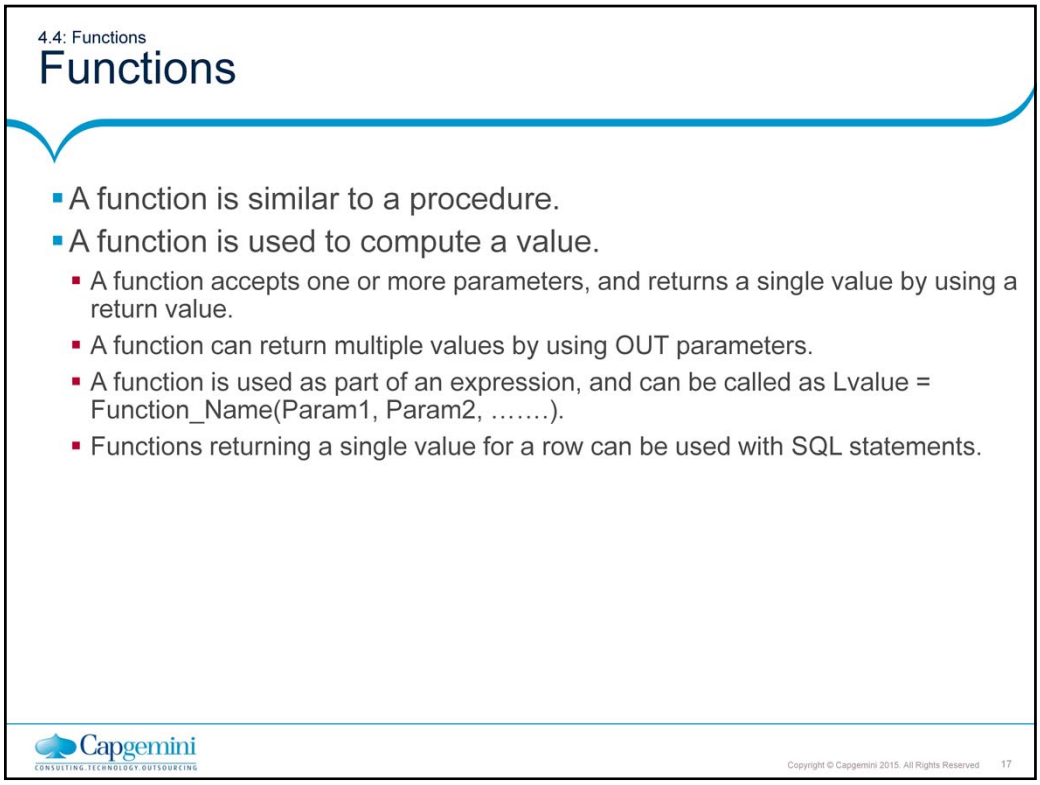
While executing the procedure, the names of the parameters must be the same as those in the procedure declaration.

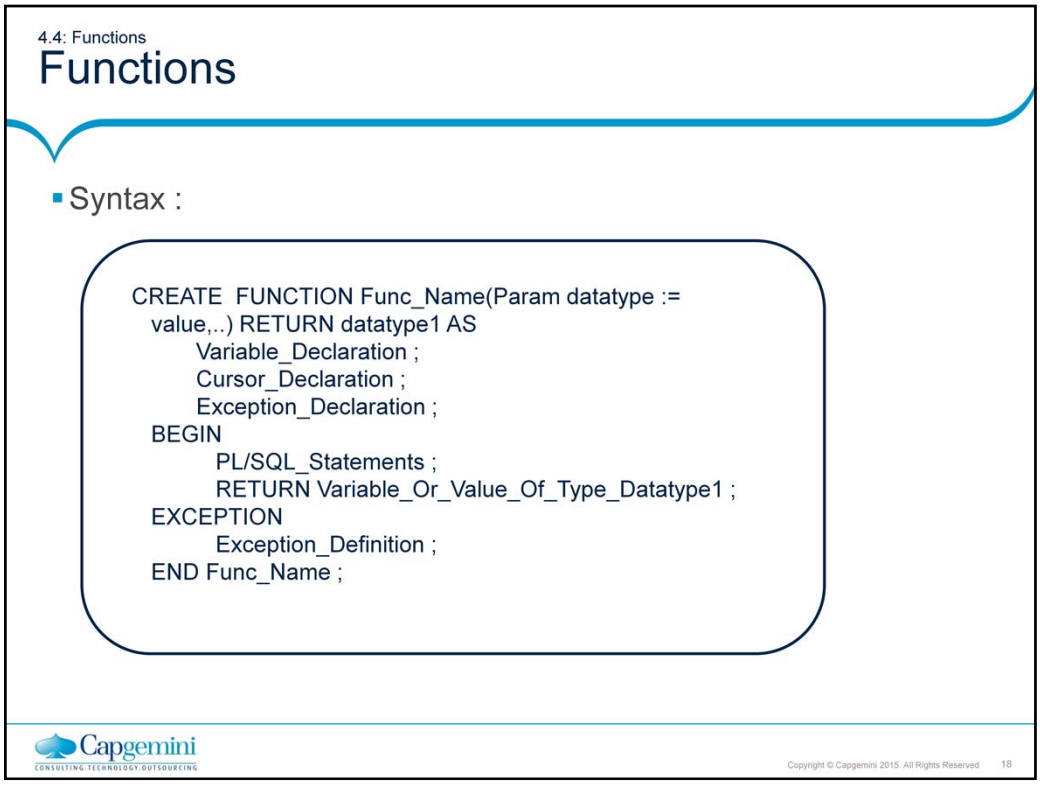


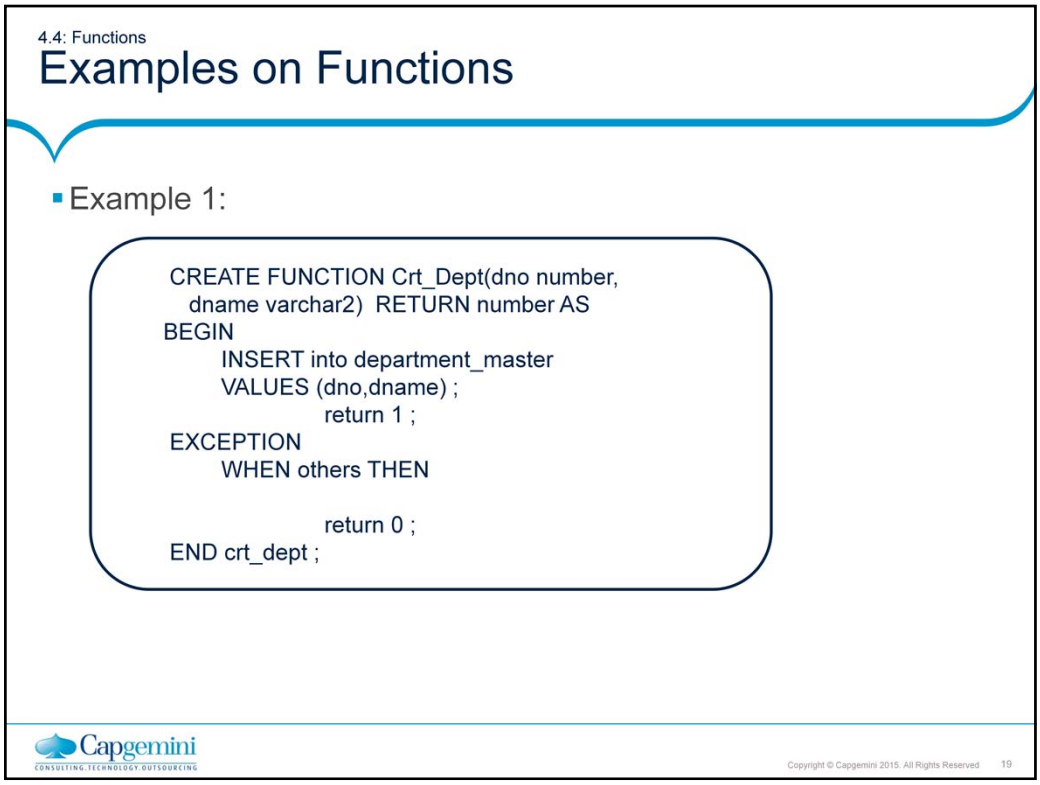

**Mixed notation:**

You specify the first parameters with “Positional notation”, and then switch to “Named notation” for the last parameters.

You can use this notation to call procedures that have some “required parameters”, followed by some “optional parameters”.


4.4: Functions
Functions
A function is similar to a procedure.
A function is used to compute a value.
A function accepts one or more parameters, and returns a single value by using a return value.
A function can return multiple values by using OUT parameters.
A function is used as part of an expression, and can be called as Lvalue = Function_Name(Param1, Param2, …….).
Functions returning a single value for a row can be used with SQL statements.
Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING
Copyright © Capgemini 2015. All Rights Reserved 17

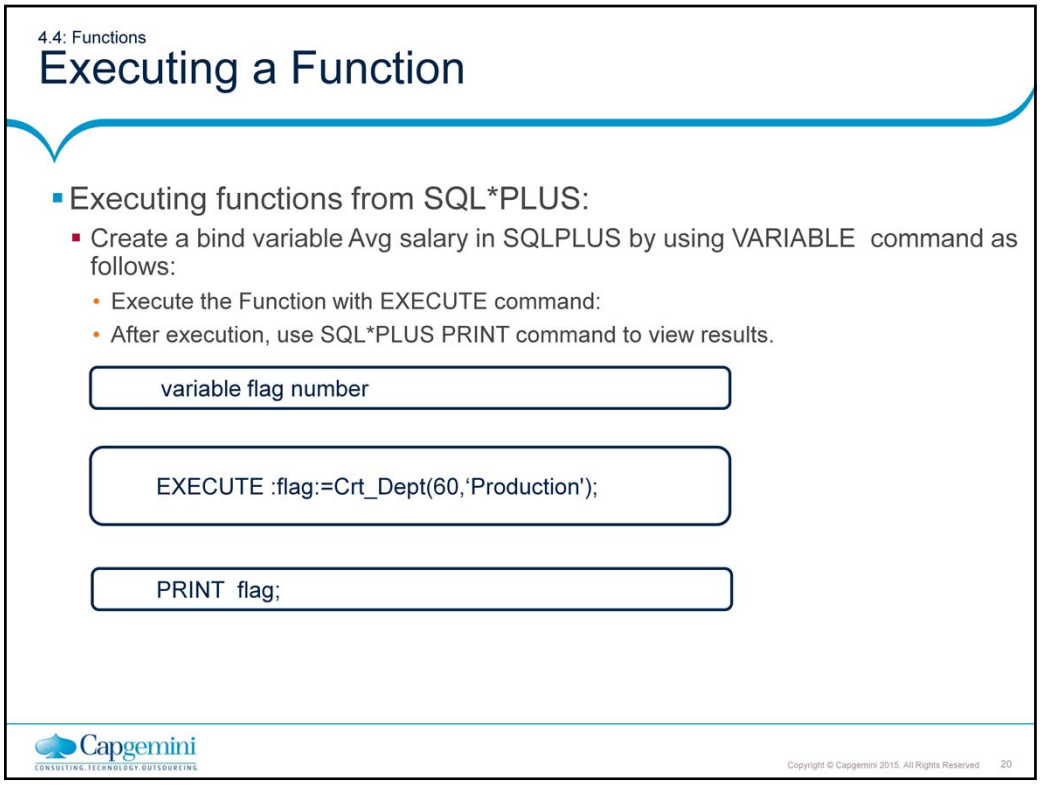

4.4: Functions
Functions
Syntax :
CREATE FUNCTION Func_Name(Param datatype :=
value,..) RETURN datatype1 AS
Variable_Declaration ;
Cursor_Declaration ;
Exception_Declaration ;
BEGIN
PL/SQL_Statements ;
RETURN Variable_Or_Value_Of_Type_Datatype1 ;
EXCEPTION
Exception_Definition ;
END Func_Name ;
Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING
Copyright © Capgemini 2015. All Rights Reserved
18




Example 2:

- Function to calculate average salary of a department:
  - Function returns average salary of the department
  - Function returns -1, in case no employees are there in the department.
  - Function returns -2, in case of any other error.

```
CREATE OR REPLACE FUNCTION Get_Avg_Sal(p_deptno
in number) RETURN number AS
   V_Sal number;
BEGIN
        SELECT Trunc(Avg(staff_sal)) INTO V_Sal
        FROM staff_master
        WHERE deptno=P_Deptno;
        IF v_sal is null THEN
                v_sal := -1 ;
         END IF;
                return v_sal;
EXCEPTION
       WHEN others THEN
              return -2;  --signifies any other errors
END get_avg_sal;
```




Functions can also be executed through command line as shown on the slide or can be called from other procedures/functions/Anonymous PL/SQL blocks.
The second snippet calls the function and passes the actual parameters. The variable declared earlier is used for collecting the return value from the function
Calling the function from an anonymous PL/SQL block

```
DECLARE
  avgsalary  number;
BEGIN
     avgsalary:=Get_Avg_Sal(20);
     dbms_output.put_line('The average salary of Dept 20 is'||
     avgsalary);
END;
```

Calling function using a Select statement

```
SELECT  Get_Avg_Sal(30) FROM staff_master;
```

## Summary

- In this lesson, you have learnt:
  - Subprograms in PL/SQL are named PL/SQL blocks.
  - There are two types of subprograms, namely: Procedures and Functions
  - Procedure is used to perform an action
    - Procedures have three subprogram parameter modes, namely: IN, OUT, and INOUT
  - Functions are used to compute a value
    - A function accepts one or more parameters, and returns a single value by using a return value
    - A function can return multiple values by using OUT parameters

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 21

## Review – Questions

- Question 1: Anonymous Blocks do not have names.
  - True / False
- Question 2: A function can return multiple values by using OUT parameters
  - True / False

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

 22

## Review – Questions

- Question 3: An ___ parameter returns a value to the caller of a subprogram.
- Question 4: A procedure contains two parts: _____ and _____.
- Question 5: In ___ notation, the order of the parameters is not significant.

# Oracle for Developers (PLSQL)

Appendices

# Appendix A.

**Built-in Packages in Oracle**

DBMS_OUTPUT: Enabling and Disabling Output

- DBMS_OUTPUT provides a mechanism for displaying information from the PL/SQL program on to your screen (that is your session's output device).
- The DBMS_OUTPUT package is created when the Oracle database is installed.
- The “dbmsoutp.sql” script contains the source code for the specification of this package.
- This script is called by the “catproc.sql” script, which is normally run immediately after database creation.
- The catproc.sql script creates the public synonym DBMS_OUTPUT for the package.
- Instance-wise access to this package is provided on installation, so no additional steps should be necessary in order to use DBMS_OUTPUT.

**DBMS_OUTPUT Program Names**

**Name:** DISABLE

**Description :** Disables output from the package; the DBMS_OUTPUT buffer will not be flushed to the screen.

**Name:** ENABLE

**Description :** Enables output from the package.

**Name:** GET_LINE

**Description :** Gets a single line from the buffer.

**Name:** GET_LINES

**Description :** Gets specified number of lines from the buffer and passes them into a PL/SQL table.

**Name:** NEW_LINE

**Description :** Inserts an end-of-line mark in the buffer.

**Name:** PUT

**Description :** Puts information into the buffer.

**Name:** PUT_LINE

**Description :** Puts information into the buffer and appends an end-of-line marker after that data.

**DBMS_OUTPUT Concepts:**

- Each user has a DBMS_OUTPUT buffer of up to 1,000,000 bytes in size. You can write information to this buffer by calling the DBMS_OUTPUT.PUT and DBMS_OUTPUT.PUT_LINE programs.
  - If you are using DBMS_OUTPUT from within SQL*Plus, this information will be automatically displayed when your program terminates.
  - You can (optionally) explicitly retrieve information from the buffer with calls to DBMS_OUTPUT.GET and DBMS_OUTPUT.GET_LINE.
- The DBMS_OUTPUT buffer can be set to a size between 2,000 and 1,000,000 bytes with the DBMS_OUTPUT.ENABLE procedure.
  - If you do not enable the package, no information will be displayed or be retrievable from the buffer.
- The buffer stores three different types of data in their internal representations, namely VARCHAR2, NUMBER, and DATE.
  - These types match the overloading available with the PUT and PUT_LINE procedures.
  - Note that DBMS_OUTPUT does not support Boolean data in either its buffer or its overloading of the PUT procedures.

**DBMS_OUTPUT Exceptions:**

- DBMS_OUTPUT does not contain any declared exceptions. Instead, Oracle designed the package to rely on two error numbers in the -20 NNN range (usually reserved for Oracle customers). You may, therefore, encounter one of these two exceptions when using the DBMS_OUTPUT package (no names are associated with these exceptions).
  - The -20000 error number indicates that these package-specific exceptions were raised by a call to RAISE_APPLICATION_ERROR, which is in the DBMS_STANDARD package.
- -20000
  - ORU-10027: buffer overflow, limit of <buf_limit> bytes.
  - If you receive the -10027 error, you should see if you can increase the size of your buffer with another call to DBMS_OUTPUT.ENABLE.
- -20000
  - ORU-10028: line length overflow, limit of 255 bytes per line.
  - If you receive the -10028 error, you should restrict the amount of data you are passing to the buffer in a single call to PUT_LINE, or in a batch of calls to PUT followed by NEW_LINE.
- You may also receive the ORA-06502 error:
  - ORA-06502
  - It is a numeric or value error.
  - If you receive the -06502 error, you have tried to pass more than 255 bytes of data to DBMS_OUTPUT.PUT_LINE. You must break up the line into more than one string.

**Drawbacks of DBMS_OUTPUT:**

- Before learning all about the DBMS_OUPUT package, and rushing to use it, you should be aware of several drawbacks with the implementation of this functionality:
  - The "put" procedures that place information in the buffer are overloaded only for strings, dates, and numbers. You cannot request the display of Booleans or any other types of data. You cannot display combinations of data (a string and a number, for instance), without performing the conversions and concatenations yourself.
  - You will see output from this package only after your program completes its execution. You cannot use DBMS_OUTPUT to examine the results of a program while it is running. And if your program terminates with an unhandled exception, you may not see anything at all!
  - If you try to display strings longer than 255 bytes, DBMS_OUTPUT will raise a VALUE_ERROR exception.
  - DBMS_OUTPUT is not a strong choice as a report generator, because it can handle a maximum of only 1,000,000 bytes of data in a session before it raises an exception.
  - If you use DBMS_OUTPUT in SQL*Plus, you may find that any leading blanks are automatically truncated. Also, attempts to display blank or NULL lines are completely ignored.
- There are workarounds for almost every one of these drawbacks. The solution invariably requires the construction of a package that encapsulates and hides DBMS_OUTPUT.

•Writing to DBMS_OUTPUT buffer

You can write information to the DBMS_OUTPUT buffer with calls to the PUT, NEW_LINE, and PUT_LINE procedures.

The DBMS_OUTPUT.PUT procedure:

The PUT procedure puts information into the buffer, but does not append a newline marker into the buffer.

Use PUT if you want to place information in the buffer (usually with more than one call to PUT), but not also automatically issue a newline marker.

- The specification for PUT is overloaded, so that you can pass data in its native format to the package without having to perform conversions.

```
PROCEDURE DBMS_OUTPUT.PUT (A VARCHAR2);
PROCEDURE DBMS_OUTPUT.PUT (A NUMBER);
PROCEDURE DBMS_OUTPUT.PUT (A DATE);
```

where A is the data being passed.

**Example:**

- In the following example, three simultaneous calls to PUT, place the employee name, department ID number, and hire date into a single line in the DBMS_OUTPUT buffer:

```
DBMS_OUTPUT.PUT (:employee.lname || ', ' ||
:employee.fname);
DBMS_OUTPUT.PUT (:employee.department_id);
DBMS_OUTPUT.PUT (:employee.hiredate);
```

- If you follow these PUT calls with a NEW_LINE call, that information can then be retrieved with a single call to GET_LINE.

The DBMS_OUTPUT.NEW_LINE procedure:

The NEW_LINE procedure inserts an end-of-line marker in the buffer.

Use NEW_LINE after one or more calls to PUT in order to terminate those entries in the buffer with a newline marker.

Given below is the specification for NEW_LINE:

```
PROCEDURE DBMS_OUTPUT.NEW_LINE
```

The DBMS_OUTPUT.PUT_LINE procedure:

The PUT_LINE procedure puts information into the buffer, and then appends a newline marker into the buffer.

The specification for PUT_LINE is overloaded, so that you can pass data in its native format to the package without having to perform conversions.

```
PROCEDURE DBMS_OUTPUT.PUT_LINE (A
VARCHAR2);
PROCEDURE DBMS_OUTPUT.PUT_LINE (A NUMBER);
PROCEDURE DBMS_OUTPUT.PUT_LINE (A DATE);
```

where A is the data being passed

- The PUT_LINE procedure is the one most commonly used in SQL*Plus to debug PL/SQL programs.
- When you use PUT_LINE in these situations, you do not need to call GET_LINE to extract the information from the buffer. Instead, SQL*Plus will automatically dump out the DBMS_OUTPUT buffer when your PL/SQL block finishes executing. (You will not see any output until the program ends.)

**Writing to DBMS_OUTPUT buffer: DBMS_OUTPUT.PUT_LINE (contd.):**
**Example:**

- Suppose that you execute the following three statements in SQL*Plus:

```
SQL> exec DBMS_OUTPUT.PUT ('I am');
SQL> exec DBMS_OUTPUT.PUT (' writing');
SQL> exec DBMS_OUTPUT.PUT ('a');
```

- You will not see anything, because PUT will place the information in the buffer, but will not append the newline marker. Now suppose you issue this next PUT_LINE command, namely:
  - SQL> exec DBMS_OUTPUT.PUT_LINE ('book!');
- Then you will see the following output:
  - I am writing a book!
- All of the information added to the buffer with the calls to PUT, patiently wait to be flushed out with the call to PUT_LINE. This is the behavior you will see when you execute individual calls at the SQL*Plus command prompt to the PUT programs.
- Suppose you place these same commands in a PL/SQL block, namely:

```
BEGIN
  DBMS_OUTPUT.PUT ('I am');
  DBMS_OUTPUT.PUT (' writing ');
  DBMS_OUTPUT.PUT ('a ');
  DBMS_OUTPUT.PUT_LINE ('book');
END;
/
```

- Then the output from this script will be exactly the same as that generated by this single call:
  - SQL> exec DBMS_OUTPUT.PUT_LINE ('I am writing a book!');

**Retrieving Data from the DBMS_OUTPUT buffer:**

You can retrieve information from the DBMS_OUTPUT buffer with call to the GET_LINE procedure.

The DBMS_OUTPUT.GET_LINE procedure:

The GET_LINE procedure retrieves one line of information from the buffer.

Given below is the specification for the procedure:

```
PROCEDURE DBMS_OUTPUT.GET_LINE    (line
OUT VARCHAR2,     status OUT INTEGER);
```

- If you are using DBMS_OUTPUT from within SQL*Plus, however, you will never need to call either of these procedures. Instead, SQL*Plus will automatically extract the information and display it on the screen for you.
- The GET_LINE procedure retrieves one line of information from the buffer.

- The parameters are summarized as shown below:

| Parameter | Description |
|---|---|
| Line | Retrieved line of text |
| Status | GET request status |

contd.

**Retrieving Data from the DBMS_OUTPUT buffer (contd.):**
**The DBMS_OUTPUT.GET_LINE procedure (contd.)**

- The line can have up to 255 bytes in it, which is not very long. If GET_LINE completes successfully, then status is set to 0. Otherwise, GET_LINE returns a status of 1.
- Note that even though the PUT and PUT_LINE procedures allow you to place information into the buffer in their native representations (dates as dates, numbers and numbers, and so forth), GET_LINE always retrieves the information into a character string. The information returned by GET_LINE is everything in the buffer up to the next newline character. This information might be the data from a single PUT_LINE or from multiple calls to PUT.
- **For example:**
  The following call to GET_LINE extracts the next line of information into a local PL/SQL variable:

```
FUNCTION get_next_line RETURN VARCHAR2
IS
  return_value VARCHAR2(255);
  get_status INTEGER;
BEGIN
  DBMS_OUTPUT.GET_LINE (return_value,
get_status);
  IF get_status = 0
  THEN
    RETURN return_value;
  ELSE
    RETURN NULL;
  END IF;
END;
```

**The UTL_FILE package**

The UTL_FILE package is created when the Oracle database is installed. The utlfile.sql script (found in the built-in packages source code directory) contains the source code for the specification of this package. This script is called by catproc.sql, which is normally run immediately after database creation. The script creates the public synonym UTL_FILE for the package and grants EXECUTE privilege on the package to public.

**Table: UTL_FILE programs**

**Name :** FCLOSE
**Description :**Closes the specified files.

**Name :** FCLOSE_ALL
**Description :** Closes all open files.

**Name :** FFLUSH
**Description :** Flushes all the data from the UTL_FILE buffer.

**Name :** FOPEN
**Description :** Opens the specified file.

**Name :** GET_LINE
**Description :** Gets the next line from the file.

**Name :** IS_OPEN
**Description :** Returns TRUE if the file is already open.

**Name :** NEW_LINE
**Description :** Inserts a newline mark in the file at the end of the current line.

**Name :** PUT
**Description :** Puts text into the buffer.

**Name :** PUT_LINE
**Description :** Puts a line of text into the file.

**Name :** PUTF
**Description :** Puts formatted text into the buffer.

**UTL_FILE: READING AND WRITING**

- UTL_FILE is a package that has been welcomed warmly by PL/SQL developers. It allows PL/SQL programs to both “read from” and “write to” any operating system files that are accessible from the server on which your database instance is running.
  - You can now read ini files and interact with the operating system a little more easily than has been possible in the past.
  - You can directly load data from files into database tables while applying the full power and flexibility of PL/SQL programming.
  - You can directly generate reports from within PL/SQL without worrying about the maximum buffer restrictions of DBMS_OUTPUT.

**File security:**

- UTL_FILE lets you read and write files accessible from the server on which your database is running. So theoretically you can use UTL_FILE to write right over your tablespace data files, control files, and so on. That is of course not a very good idea.
- Server security requires the ability to place restrictions on where you can read and write your files.
- UTL_FILE implements this security by limiting access to files that reside in one of the directories specified in the INIT.ORA file for the database instance on which UTL_FILE is running.
- When you call FOPEN to open a file, you must specify both the location and the name of the file, in separate arguments. This file location is then checked against the list of accessible directories.
- Given below is the format of the parameter for file access in the INIT.ORA file:
  - utl_file_dir = <directory>
- Include a parameter for utl_file_dir for each directory you want to make accessible for UTL_FILE operations. For example: The following entries enable four different directories in UNIX:
  - utl_file_dir = /tmp
  - utl_file_dir = /ora_apps/hr/time_reporting
  - utl_file_dir = /ora_apps/hr/time_reporting/log
  - utl_file_dir = /users/test_area
- To bypass server security and allow read/write access to all directories, you can use the special syntax given below:
  - utl_file_dir = *

**Specifying file locations:**

- The location of the file is an operating system-specific string that specifies the directory or area in which to open the file. The location you provide must have been listed as an accessible directory in the INIT.ORA file for the database instance.
- The INIT.ORA location is a valid directory or area specification, as shown in the following examples:
  - In Windows NT:
  - 'k:\common\debug'
  - In UNIX:
  - '/usr/od2000/admin'
- Few examples are given below:
  - In Windows NT: file_id := UTL_FILE.FOPEN ('k:\common\debug', 'trace.lis', 'R');
  - In UNIX: file_id := UTL_FILE.FOPEN ('/usr/od2000/admin', 'trace.lis', 'W');
- Your location must be an explicit, complete path to the file. You cannot use operating system-specific parameters such as environment variables in UNIX to specify file locations.

**UTL_FILE exceptions:**

- The package specification of UTL_FILE defines seven exceptions. The cause behind a UTL_FILE exception can often be difficult to understand.
- Given below are the explanations Oracle provides for each of the exceptions:
  - **INVALID_PATH**
    The file location or the filename is invalid. Perhaps the directory is not listed as a utl_file_dir parameter in the INIT.ORA file (or doesn't exist as all), or you are trying to read a file and it does not exist.
  - **INVALID_MODE**
    The value you provided for the open_mode parameter in UTL_FILE.FOPEN was invalid. It must be “A”, “R”, or “W”.
  - **INVALID_FILEHANDLE**
    The file handle you passed to a UTL_FILE program was invalid. You must call UTL_FILE.FOPEN to obtain a valid file handle.
  - **INVALID_OPERATION**
    UTL_FILE could not open or operate on the file as requested. For example, if you try to write to a read-only file, you will raise this exception.
  - **READ_ERROR**
    The operating system returned an error when you tried to read from the file. (This does not occur very often.)
  - **WRITE_ERROR**
    The operating system returned an error when you tried to write to the file. (This does not occur very often.)
  - **INTERNAL_ERROR**
    Uh-oh. Something went wrong and the PL/SQL runtime engine couldn't assign blame to any of the previous exceptions. Better call Oracle Support!
- Programs in UTL_FILE may also raise the following standard system exceptions:
  - **NO_DATA_FOUND**
    It is raised when you read past the end of the file with UTL_FILE.GET_LINE.
  - **VALUE_ERROR**
    It is raised when you try to read or write lines in the file which are too long.
  - **INVALID_MAXLINESIZE**
    Oracle 8.0 and above: It is raised when you try to open a file with a maximum linesize outside of the valid range (between 1 through 32767).

You can use the FOPEN and IS_OPEN functions when you open files via UTL_FILE.

Note:

Using the UTL-FILE package, you can only open a maximum of ten files for each Oracle session.

UTL_FILE provides only one program to retrieve data from a file, namely the GET_LINE procedure.

UTL_FILE.FOPEN function:

The FOPEN function opens the specified file and returns a file handle that you can then use to manipulate the file.

The header for the function is:

```
FUNCTION UTL_FILE.FOPEN (FUNCTION UTL_FILE.FOPEN
(location IN VARCHAR2, location IN VARCHAR2, filename IN
VARCHAR2, filename IN VARCHAR2, open_mode IN VARCHAR2)
open_mode IN VARCHAR2, RETURN file_type; max_linesize IN
BINARY_INTEGER) RETURN file_type;
```

**UTL_FILE.FOPEN Function:**

- Parameters for the function shown in the slide are summarized in the following table.

**Parameter :** location

**Description** : Location of the file

**Parameter :** filename

**Description :** Name of the file

**Parameter :** openmode

**Description :** Mode in which the file has to be opened

**Parameter :** max_linesize

**Description :** The maximum number of characters per line, including the newline character, for this file. Minimum is 1, maximum is 32767.

contd.

**UTL_FILE.FOPEN Function (contd.):**

- You can open the file in one of the following three modes:
  - **R mode**
    Open the file read-only. If you use this mode, use UTL_FILE's GET_LINE procedure to read from the file.
  - **W mode**
    Open the file to read and write in replace mode. When you open in replace mode, all existing lines in the file are removed. If you use this mode, then you can use any of the following UTL_FILE programs to modify the file: PUT, PUT_LINE, NEW_LINE, PUTF, and FFLUSH.
  - **A mode**
    Open the file to read and write in append mode. When you open in append mode, all existing lines in the file are kept intact. New lines will be appended after the last line in the file. If you use this mode, then you can use any of the following UTL_FILE programs to modify the file: PUT, PUT_LINE, NEW_LINE, PUTF, and fFFLUSH.
- Example
  - The following example shows how to declare a file handle, and then open a configuration file for that handle in read-only mode:

```
DECLARE
        config_file UTL_FILE.FILE_TYPE;
BEGIN
        config_file := UTL_FILE.FOPEN ('/maint/admin',
'config.txt', 'R');
```

UTL_FILE.IS_OPEN function:

The IS_OPEN function returns TRUE if the specified handle points to a file that is already open. Otherwise, it returns FALSE.

The header for the function is:

where file is the file to be checked.

```
FUNCTION UTL_FILE.IS_OPEN (file IN
UTL_FILE.FILE_TYPE) RETURN BOOLEAN;
```

**Reading from Files**

**UTL_FILE.GET_LINE procedure:**

- The GET_LINE procedure reads a line of data from the specified file, if it is open, into the provided line buffer. Given below is the header for the procedure:

```
PROCEDURE UTL_FILE.GET_LINE
  (file IN UTL_FILE.FILE_TYPE,
   buffer OUT VARCHAR2);
```

- Parameters are summarized in the following table:

**Parameter :** File

**Description :**The file handle returned by a call to FOPEN.

**Parameter :** Buffer

**Description :** The buffer into which the line of data is read.

contd.

**UTL_FILE.GET_LINE procedure (contd.):**

- The variable specified for the buffer parameter must be large enough to hold all the data up to the next carriage return or end-of-file condition in the file. If not, PL/SQL will raise the VALUE_ERROR exception. The line terminator character is not included in the string passed into the buffer.
- For example:
  Since GET_LINE reads data only into a string variable, you will have to perform your own conversions to local variables of the appropriate datatype if your file holds numbers or dates. Of course, you can call this procedure and directly read data into string and numeric variables, as well. In this case, PL/SQL will be performing a runtime, implicit conversion for you. In many situations, this is fine.
- It is generally recommended that you avoid implicit conversions and instead perform your own conversion. This approach more clearly documents the steps and dependencies.
- Here is an example:

```
DECLARE
  fileID UTL_FILE.FILE_TYPE;
  strbuffer VARCHAR2(100);
  mynum NUMBER;
BEGIN
  fileID := UTL_FILE.FOPEN ('/tmp', 'numlist.txt', 'R');
  UTL_FILE.GET_LINE (fileID, strbuffer);
  mynum := TO_NUMBER (strbuffer);
END;
/
```

- When GET_LINE attempts to read past the end of the file, the NO_DATA_FOUND exception is raised. This is the same exception that is raised when you:
  - execute an implicit (SELECT INTO) cursor that returns no rows, or
  - reference an undefined row of a PL/SQL (nested in PL/SQL8) table.
- If you are performing more than one of these operations in the same PL/SQL block, remember that this same exception can be caused by very different parts of your program.

**Writing to Files: UTL_FILE.PUT procedure**

- The PUT procedure puts data out to the specified open file.
- Given below is the header for this procedure:

```
PROCEDURE UTL_FILE.PUT
   (file IN UTL_FILE.FILE_TYPE,
   buffer IN VARCHAR2);      ----------OUT  replace with
IN
```

- Parameters are summarized in the following table.

| Parameter | Description |
|---|---|
| File | The file handle returned by a call to FOPEN. |
| Buffer | The buffer containing the text to be written to the file; maximum size allowed is 32K for Oracle 8.0.3 and above; for earlier versions, it is 1023 bytes. |

- The PUT procedure adds the data to the current line in the opened file, but does not append a line terminator. You must use the NEW_LINE procedure to terminate the current line or use PUT_LINE to write out a complete line with a line termination character.

**Exceptions:**

- PUT may raise any of the following exceptions:
  - UTL_FILE.INVALID_FILEHANDLE
  - UTL_FILE.INVALID_OPERATION
  - UTL_FILE.WRITE_ERROR

UTL_FILE offers a number of different procedures you can use to write to a file:

UTL_FILE.PUT procedure

Puts a piece of data (string, number, or date) into a file in the current line.

UTL_FILE.NEW_LINE procedure

Puts a newline or line termination character into the file at the current position.

UTL_FILE.PUT_LINE procedure

Puts a string into a file, followed by a platform-specific line termination character.

UTL_FILE.PUTF procedure

Puts up to five strings out to the file in a format based on a template string, similar to the printf function in C.

**UTL_FILE.PUT_LINE procedure:**

- This procedure writes data to a file, and then immediately appends a newline character after the text. Given below is the header for PUT_LINE:

Parameters are summarized in the following table.

```
PROCEDURE UTL_FILE.PUT_LINE
   (file IN UTL_FILE.FILE_TYPE,
   buffer IN VARCHAR2);
```

| Parameter | Description |
|---|---|
| File | The file handle returned by a call to FOPEN. |
| Buffer | Text to be written to the file; maximum size allowed is 32K for Oracle 8.0. 3 and above; for earlier versions, it is 1023 bytes. |

- Before you can call UTL_FILE.PUT_LINE, you must have already opened the file.

contd.

**UTL_FILE.FCLOSE procedure:**

- Use FCLOSE to close an open file. The header for this procedure is:

  where file is the file handle.

- Note that the argument to UTL_FILE.FCLOSE is an IN OUT parameter, because the procedure sets the id field of the record to NULL after the file is closed.
- If there is buffered data that has not yet been written to the file when you try to close it, UTL_FILE will raise the WRITE_ERROR exception.

```
PROCEDURE UTL_FILE.FCLOSE (file IN OUT
FILE_TYPE);
```

**UTL_FILE.FCLOSE_ALL procedure:**

- FCLOSE_ALL closes all the opened files. Given below is the header for this procedure:

```
PROCEDURE UTL_FILE.FCLOSE_ALL;
```

- This procedure will come in handy when you have opened a variety of files and want to make sure that none of them are left open when your program terminates.
- In programs in which files have been opened, you should also call FCLOSE_ALL in exception handlers in programs. If there is an abnormal termination of the program, files will then still be closed.

```
EXCEPTION
  WHEN OTHERS

THEN
    UTL_FILE.FCLOSE_ALL;
    ... other clean up activities ...
END;
```

- NOTE: When you close your files with the FCLOSE_ALL procedure, none of your file handles will be marked as closed (the id field, in other words, will still be non-NULL). The result is that any calls to IS_OPEN for those file handles will still return TRUE. You will not, however, be able to perform any read or write operations on those files (unless you reopen them).

**Exceptions**

- FCLOSE_ALL may raise the exception UTL_FILE.WRITE_ERROR.

**Handling LOB (Large Objects):**

- Large Objects (LOBs) are a set of datatypes that are designed to hold large amounts of data.

  Two types of LOBs are supported:

  Those stored in the database either in-line in the table or in a separate segment or tablespace, such as BLOB, CLOB, and NCLOB. LOBs in the database are stored inside database tablespaces in a way that optimizes space and provides efficient access. The following SQL datatypes are supported for declaring internal LOBs: BLOB, CLOB, and NCLOB

  Those stored as operating system files, such as BFILEs

- LOBs are designed to support Unstructured kind of data.

**Unstructured Data:**

- Unstructured Data cannot be decomposed into Standard Components:
  - Unstructured data cannot be decomposed into standard components. Data about an Employee can be “structured” into a Name (probably a character string), an identification (likely a number), a Salary, and so on. But if you are given a Photo, you find that the data really consists of a long stream of 0s and 1s. These 0s and 1s are used to switch pixels on or off so that you will see the Photo on a display. However, they cannot be broken down into any finer structure in terms of database storage.
- Unstructured Data is Large:
  - Also interesting is the fact that unstructured data such as text, graphic images, still video clips, full motion video, and sound waveforms tend to be large - a typical employee record may be a few hundred bytes, but even small amounts of multimedia data can be thousands of times larger.
- Unstructured Data in System Files needs Accessing from the Database:
  - Finally, some multimedia data may reside on operating system files, and it is desirable to access them from the database.

contd.

- Given below is a summary of all the four types of LOBs :
  - BLOB (Binary LOB)
    BLOB stores unstructured binary data up to 4GB in length.
    For example: Video or picture information
  - CLOB (Character LOB)
    CLOB stores single-byte character data up to 4GB in length.
    For example: Store document
  - NCLOB (National CLOB)
    NCLOB stores a CLOB column that supports multi-byte characters from National Character set defined by Oracle 8 database.
  - BFILE (Binary File)
    BFILE stores read-only binary data as an external file outside the database.
    Internal objects store a locator in the Large Object column of a table.
    Locator is a pointer that specifies the actual location of LOB stored out-of-line.
    The LOB locator for BFILE is the pointer to the location of the binary file stored in the operating system.
    Oracle supports data integrity and concurrency for all the LOBs except for BFILEs.

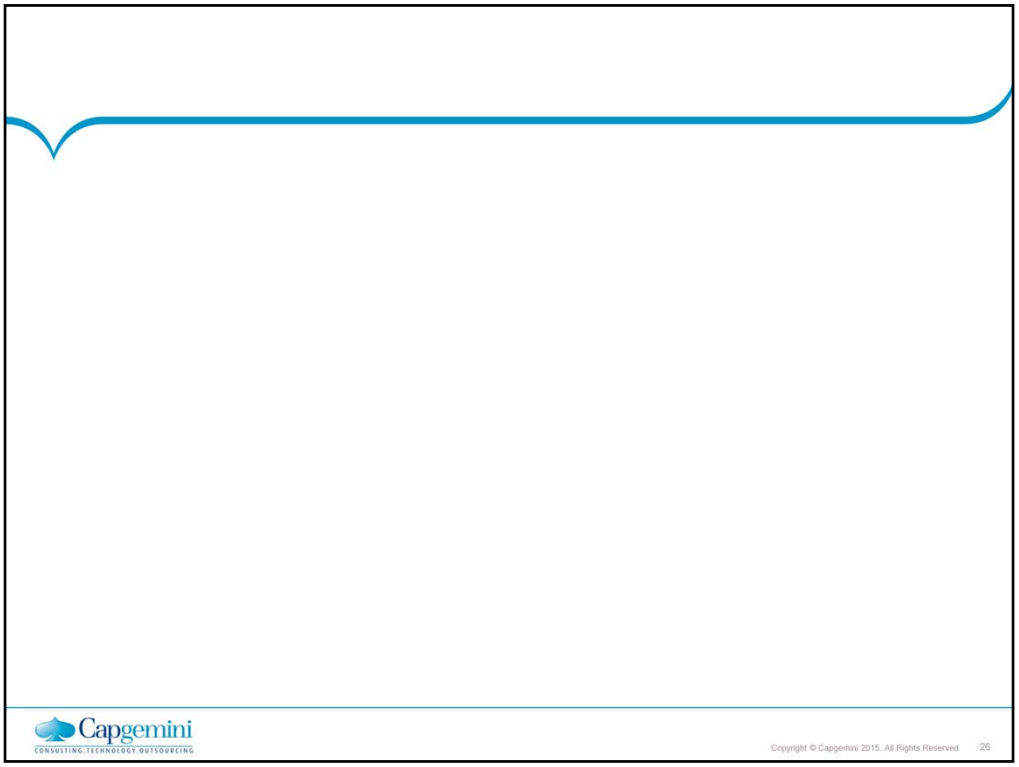
**Handling LOB (Large Objects) (contd.):**
**Unstructured Data (contd.):**

- LOB Datatype helps support Internet Applications:
  - With the growth of the internet and content-rich applications, it has become imperative that the database supports a datatype that fulfills the following:
    - datatype should store unstructured data
    - datatype should be optimized for large amounts of such data
    - datatype should provide an uniform way of accessing large unstructured data within the database or outside

**BFILE considerations:**

- There are some special considerations you should be aware of when you work with BFILEs.
  - **The DIRECTORY object**
    - A BFILE locator consists of a directory alias and a filename. The directory alias is an Oracle8 database object that allows references to operating system directories without hard-coding directory pathnames. This statement creates a directory:
      - → CREATE DIRECTORY IMAGES AS 'c:\images';
    - To refer to the c:\images directory within SQL, you can use the IMAGES alias, rather than hard-coding the actual directory pathname.
    - To create a directory, you need the CREATE DIRECTORY or CREATE ANY DIRECTORY privilege. To reference a directory, you must be granted the READ privilege, as in:
      - → GRANT READ ON DIRECTORY IMAGES TO SCOTT;
  - **Populating a BFILE locator**
    - The Oracle8 built-in function BFILENAME can be used to populate a BFILE locator. BFILENAME is passed a directory alias and filename and returns a locator to the file. In the following block, the BFILE variable corporate_logo is assigned a locator for the file named ourlogo.bmp located in the IMAGES directory:

- Once a BFILE column or variable is associated with a physical file, read operations on the BFILE can be performed using the DBMS_LOB package.
- Remember that access to physical files via BFILEs is read-only, and that the BFILE value is a pointer. The contents of the file remain outside of the database, but on the same server on which the database resides.




```
DECLARE
  corporate_logo     BFILE;
BEGIN
  corporate_logo := BFILENAME ( 'IMAGES',
'ourlogo.bmp' );
END;
The following statements populate the my_book_files
table; each row is associated with a file in the
BOOK_TEXT directory:
INSERT INTO my_book_files ( file_descr, book_file )
    VALUES ( 'Chapter 1', BFILENAME('BOOK_TEXT',
'chapter01.txt') );
UPDATE  my_book_files
  SET  book_file = BFILENAME( 'BOOK_TEXT',
'chapter02rev.txt' )
 WHERE  file_descr = 'Chapter 2';
```

**Internal LOB considerations: Few more points:**

Given below are a few more special considerations for Internal LOBs.

- **Retaining the LOB locator**
  - The following statement populates the my_book_text table, which contains CLOB column chapter_text:
  - Programs within the DBMS_LOB package require a LOB locator to be passed as input. If you want to insert the preceding row and then call a DBMS_LOB program using the row's CLOB value, you must retain the LOB locator created by your INSERT statement.
  - You can do this as shown in the following block, which inserts a row, selects the inserted LOB locator, and then calls the DBMS_LOB.GETLENGTH program to get the size of the CLOB chapter_text column. Note that the GETLENGTH program expects a LOB locator.

```
INSERT INTO my_book_text ( chapter_descr, chapter_text )
VALUES ( 'Chapter 1', 'It was a dark and stormy night.' );
```

```
DECLARE
  chapter_loc         CLOB;
  chapter_length      INTEGER;
BEGIN
  INSERT INTO my_book_text ( chapter_descr,
chapter_text )
      VALUES ( 'Chapter 1', 'It was a dark and stormy
night.' );
  SELECT  chapter_text
    INTO  chapter_loc
    FROM  my_book_text
   WHERE  chapter_descr = 'Chapter 1';
  chapter_length := DBMS_LOB.GETLENGTH(
chapter_loc );
  DBMS_OUTPUT.PUT_LINE( 'Length of Chapter 1: ' ||
chapter_length );
END;
/
```

This is the output of the script:

```
Length of Chapter 1: 31
```

contd.

**Internal LOB considerations: Few more points (contd.):**

- **The RETURNING clause**
  - You can avoid the second trip to the database (i.e. the SELECT of the LOB locator after the INSERT) by using a RETURNING clause in the INSERT statement.
  - By using this feature, perform the INSERT operation and the LOB locator value for the new row in a single operation.

```
DECLARE
  chapter_loc       CLOB;
  chapter_length    INTEGER;
BEGIN

  INSERT INTO my_book_text ( chapter_descr, chapter_text
)
      VALUES ( 'Chapter 1', 'It was a dark and stormy night.' )
   RETURNING chapter_text INTO chapter_loc;

  chapter_length := DBMS_LOB.GETLENGTH( chapter_loc
);

  DBMS_OUTPUT.PUT_LINE( 'Length of Chapter 1: ' ||
chapter_length );

END;
/
```

This is the output of the script:
Length of Chapter 1: 31

  - The RETURNING clause can be used in both INSERT and UPDATE statements.

contd.

**Internal LOB considerations: Few more points (contd.):**

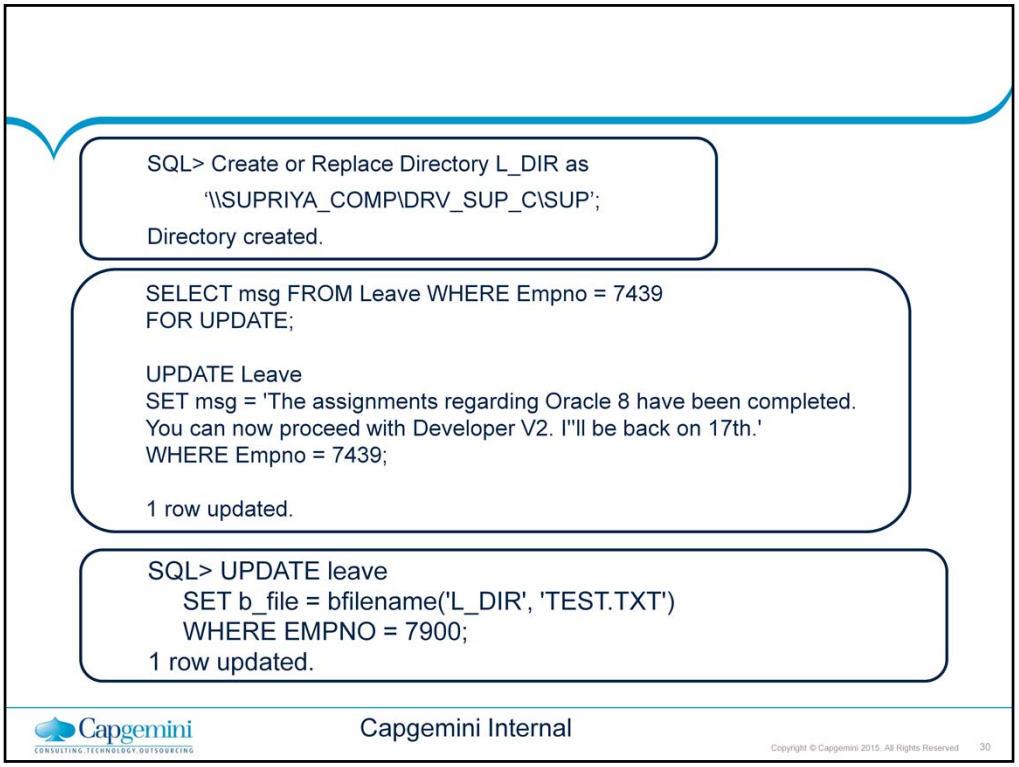
- **NULL versus "empty" LOB locators**
  - Oracle8 provides the built-in functions EMPTY_BLOB and EMPTY_CLOB to set BLOB, CLOB, and NCLOB columns to "empty".
  - For example:

  - The LOB data is set to NULL. However, the associated LOB locator is assigned a valid locator value, which points to the NULL data. This LOB locator can then be passed to DBMS_LOB programs.

```
INSERT INTO my_book_text ( chapter_descr, chapter_text )
VALUES ( 'Table of Contents', EMPTY_CLOB() );
```

This is the output of the script:
Length of Table of Contents: 0

  - Note that EMPTY_CLOB can be used to populate both CLOB and NCLOB columns. EMPTY_BLOB and EMPTY_CLOB can be called with or without empty parentheses.
  - Note: Do not populate BLOB, CLOB, or NCLOB columns with NULL values. Instead, use the EMPTY_BLOB or EMPTY_CLOB functions, which will populate the columns with a valid LOB locator and set the associated data to NULL.




Accessing External LOBs - Examples**Example 1:** Create a directory object as shown below:

**Example 2:** In the following statement, we associate the file TEST.TXT for Empno 7900.

Read operations on the BFILE can be performed by using PL/SQL DBMS_LOB package and OCI.

These files are read-only through BFILES.

These files cannot be updated or deleted through BFILES

While updating LOBs:

Explicitly lock the rows.

Use the FOR UPDATE clause in a SELECT statement.

- To update a column that uses the BFILE datatype, you do not have to lock the rows.
- Using a DBMS_LOB Package:
  - Provides procedures to access LOBs.
  - Allows reading and modifying BLOBs, CLOBs, and NCLOBs, and provides read-only operations on BFILEs.
- All DBMS_LOB routines work based on LOB locators.

**DBMS_LOB Package Routines:**

The routines that can modify **BLOB**, **CLOB**, and **NCLOB** values are:

- APPEND() - appends the contents of the source LOB to the destination LOB
- COPY() - copies all or part of the source LOB to the destination LOB
- ERASE() - erases all or part of a LOB
- LOADFROMFILE() - loads BFILE data into an internal LOB
- TRIM() - trims the LOB value to the specified shorter length
- WRITE() - writes data to the LOB from a specified offset

The routines that read or examine **LOB** values are:

- GETLENGTH() - gets the length of the LOB value
- INSTR() - returns the matching position of the nth occurrence of the pattern in the LOB
- READ() - reads data from the LOB starting at the specified offset
- SUBSTR() - returns part of the LOB value starting at the specified offset

The read-only routines specific to **BFILEs** are:

- FILECLOSE() - closes the file
- FILECLOSEALL()- closes all previously opened files
- FILEEXISTS() - checks if the file exists on the server
- FILEGETNAME() - gets the directory alias and file name
- FILEISOPEN() - checks if the file was opened using the input BFILE
- locators
- FILEOPEN() - opens a file

**DBMS_LOB Exceptions:**
A DBMS_LOB function or procedure can raise any of the named exceptions shown in the table.

| Exception | Code in error msg | Meaning |
|---|---|---|
| INVALID_ARGVAL | 21560 | “argument %s is null, invalid, or out of range” |
| ACCESS_ERROR | 22925 | Attempt to read/write beyond maximum LOB size on <n>. |
| NO_DATA_FOUND | 1403 | EndofLOB indicator for looping read operations. |
| VALUE_ERROR | 6502 | Invalid value in parameter. |

# Oracle PL/SQL

# Lab Book

# Document Revision History

| Date | Revision No. | Author | Summary of Changes |
|---|---|---|---|
| 05-Feb-2009 | 0.1D | Rajita Dhumal | Content Creation |
| 09-Feb-2009 | | CLS team | Review |
| 02-Jun-2011 | 2.0 | Anu Mitra | Integration Refinements |
| 30-Nov-2012 | 3.0 | HareshkumarChandiramani | Revamp of Assignments and Conversion to iGATE format. |
| 22-Apr--2015 | 4.0 | Kavita Arora | Rearranging the lab questions |
| 9-May-2016 | 5.0 | Kavita Arora | Integration Refinements |

# Table of Contents



# Getting Started

**Overview**

This lab book is a guided tour for learning Oracle 9i. It comprises 'To Do' assignments. Follow the steps provided and work out the 'To Do' assignments.

**Setup Checklist for Oracle 9i**

Here is what is expected on your machine in order for the lab to work.

**Minimum System Requirements**

- Intel Pentium 90 or higher (P166 recommended)
- Microsoft Windows 95, 98, or NT 4.0, 2k, XP,7.
- Memory: 32MB of RAM (64MB or more recommended)

**Please ensure that the following is done:**

- Oracle Client is installed on every machine
- Connectivity to Oracle Server

**Instructions**

- For all coding standards refer Appendix A. All lab assignments should refer coding standards.
- Create a directory by your name in drive <drive>. In this directory, create a subdirectory Oracle 9i_assgn. For each lab exercise create a directory as lab <lab number>.

**Learning More (Bibliography if applicable)**

- Oracle10g - SQL - Student Guide - Volume 1 by Oracle Press
- Oracle10g - SQL - Student Guide - Volume 2 by Oracle Press
- Oracle10g database administration fundamentals volume 1 by Oracle Press
- Oracle10g Complete Reference by Oracle Press
- Oracle10g SQL with an Introduction to PL/SQL by Lannes L. Morris-Murphy

# Lab 1.Introduction to PL/SQL and Cursors

| | |
|---|---|
| **Goals** | • The following set of exercises are designed to implement the following<br>• PL/SQL variables and data types<br>• Create, Compile and Run anonymous PL/SQL blocks<br>• Usage of Cursors |
| **Time** | 1hr |

**1.1 Identify the problems(if any) in the below declarations:**

```
DECLARE
V_Sample1 NUMBER(2);
V_Sample2 CONSTANT NUMBER(2) ;
V_Sample3 NUMBER(2) NOT NULL ;
V_Sample4 NUMBER(2) := 50;
V_Sample5 NUMBER(2) DEFAULT 25;
```

**Example 1: Declaration Block**

**1.2 The following PL/SQL block is incomplete.**

Modify the block to achieve requirements as stated in the comments in the block.

```
DECLARE --outer block
var_num1 NUMBER := 5;
BEGIN
DECLARE --inner block
var_num1 NUMBER := 10;
BEGIN
DBMS_OUTPUT.PUT_LINE('Value for var_num1:' ||var_num1);
--Can outer block variable (var_num1) be printed here.IfYes,Print the same.
END;
--Can inner block variable(var_num1)  be printed here.IfYes,Print the same.
END;
```

**Example 2: PL/SQL block**

**1.3 Write a PL/SQL program**

Write a PL/SQL program to display the details of the employee number 7369.

**1.4 Write a PL/SQL program**

Write a PL/SQL program to accept the Employee Name and display the details of that Employee including the Department Name.

**1.5.Write a PL/SQL block to increase the salary of employees**

Write a PL/SQL block to increase the salary of employees either by 30 % or 5000 whichever is minimum for a given Department_Code.

Find out 30% of salary, if it is more than 5000, increase by 5000. If it is less than 5000, increase by 30% of salary

## Lab 2.Lab 2.Exception Handling

| | |
|---|---|
| **Goals** | Implementing Exception Handling ,Analyzing and Debugging |
| **Time** | 30 mins |

**2.1 The following PL/SQL block attempts to calculate bonus of staff.**

The following PL/SQL block attempts to calculate bonus of staff for a given MGR_CODE. Bonus is to be considered as twice of salary. Though Exception Handling has been implemented but block is unable to handle the same.

Debug and verify the current behavior to trace the problem.

```
DECLARE
V_BONUS V_SAL%TYPE;
V_SAL STAFF_MASTER.STAFF_SAL%TYPE;

BEGIN
SELECT STAFF_SAL INTO V_SAL
FROM STAFF_MASTER
WHERE MGR_CODE=100006;

V_BONUS:=2*V_SAL;
DBMS_OUTPUT.PUT_LINE('STAFF SALARY IS ' || V_SAL);
DBMS_OUTPUT.PUT_LINE('STAFF BONUS IS ' || V_BONUS);

EXCEPTION
WHEN NO_DATA_FOUND THEN
DBMS_OUTPUT.PUT_LINE('GIVEN CODE IS NOT VALID.ENTER VALID CODE');
END;
```

**Example 3: PL/SQL block**

**2.2 Rewrite the above block.**

Rewrite the above block to achieve the requirement.

**2.3: Write a PL/SQL program**

Write a PL/SQL program to check for the commission for an employee no 7369. If no commission exists, then display the error message. Use Exceptions.

## Lab 3.Database Programming

| | |
|---|---|
| **Goals** | • The following set of exercises are designed to implement the following<br>• Implement business logic using Database Programming like Procedures and Functions<br>• Implement validations in Procedures and Functions |
| **Time** | 2 Hrs |

**Note:** Procedures and functions should handle validations, pre-defined oracle server and user defined exceptions wherever applicable. Also use cursors wherever applicable.

**3.1. Write a function to compute age.**

The function should accept a date and return age in years.

**3.2 Write a procedure to find the manager of a staff.**

Procedure should return the following – Staff_Code, Staff_Name, Dept_Code and Manager Name.

**3.3. Write a function to compute the following.**

Function should take Staff_Code and return the cost to company.

DA = 15% Salary, HRA= 20% of Salary, TA= 8% of Salary.

Special Allowance will be decided based on the service in the company.

| | |
|---|---|
| < 1 Year | Nil |
| >=1 Year< 2 Year | 10% of Salary |
| >=2 Year< 4 Year | 20% of Salary |
| >4 Year | 30% of Salary |

**3.4. Write a procedure that accept Staff_Code**

Write a procedure that accept Staff_Code and update the salary and store the old salary details in Staff_Master_Back (Staff_Master_Back has the same structure without any constraint) table.

Exp< 2 then no Update

Exp> 2 and < 5 then 20% of salary

Exp> 5 then 25% of salary

**3.5. Write a procedure to insert details into Book_Transaction table.**

Procedure should accept the book code and staff/student code. Date of issue is current date and the expected return date should be 10 days from the current date. If the expected return date falls on Saturday or Sunday, then it should be the next working day

# Appendices

### Appendix A: Oracle Standards

Key points to keep in mind:

1. Write comments in your stored Procedures, Functions and SQL batches generously, whenever something is not very obvious. This helps other programmers to clearly understand your code. Do not worry about the length of the comments, as it will not impact the performance.
2. Prefix the table names with owner names, as this improves readability, and avoids any unnecessary confusion.

**Some more Oracle standards:**

To be shared by Faculty in class

**Appendix B: Coding Best Practices**

1. Perform all your referential integrity checks and data validations by using constraints (foreign key and check constraints). These constraints are faster than triggers. So use triggers only for auditing, custom tasks, and validations that cannot be performed by using these constraints.
2. Do not call functions repeatedly within your stored procedures, triggers, functions, and batches. For example: You might need the length of a string variable in many places of your procedure. However do not call the LENGTH function whenever it is needed. Instead call the LENGTH function once, and store the result in a variable, for later use.

## Appendix C: Table of Examples

