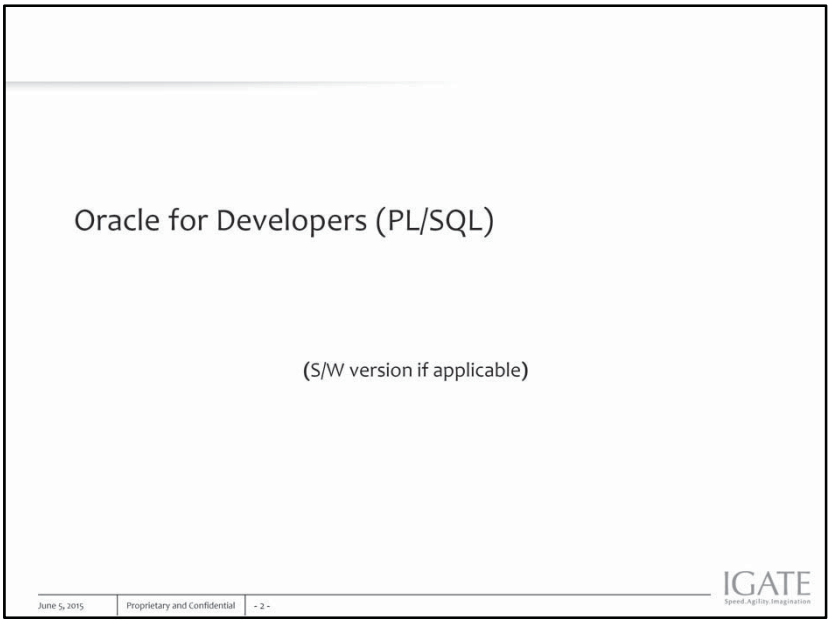
Oracle for Developers (PL/SQL)


Oracle for Developers (PL/SQL)
(S/W version if applicable)
June 5, 2015
Proprietary and Confidential
- 2 -
IGATE
Speed.Agility.Imagination

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

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 Packages, Procedures and Functions.
  - To implement Complex Business Rule using Triggers, Constraints.
- **Course Non Goals**
  - Object Oriented programming concepts (ORDBMS) are not covered as a part of this course.

Pre-requisites

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

Intended Audience

- **Software Programmers**
- **Software Analysts**

Day Wise Schedule

- **Day 1**
  - Lesson 1: PL/SQL Basics
  - Lesson 2: Introduction to Cursors
- **Day 2**
  - Lesson 2: Introduction to Cursors (Cursor variables)
  - Lesson 3: Exception Handling
  - Lesson 4: Procedures, Functions and Packages
- **Day 3**
  - Lesson 5: Built-in packages in Oracle
  - Lesson 6: Database Triggers

Table of Contents

- **Lesson 1: PL/SQL Basics**
  - 1.1: Introduction to PL/SQL
  - 1.2: PL/SQL Block Structure
  - 1.3: Handling Variables in PL/SQL
  - 1.4: Declaring a PL/SQL table
  - 1.5: Scope and Visibility of Variables
  - 1.6: SQL in PL/SQL
  - 1.7: Programmatic Constructs

Table of Contents

- **Lesson 2: Introduction to Cursors**
  - 2.1: Introduction to Cursors
  - 2.2: Implicit Cursors
  - 2.3: Explicit Cursors
  - 2.4: Cursor Attributes
  - 2.5: Processing Implicit Cursors and Explicit Cursors
  - 2.6: Cursor with Parameters
  - 2.7: Usage of Cursor Variables
  - 2.8: Difference between Cursors and Cursor Variables
- **Lesson 3: Exception Handling**
  - 3.1: Error Handling (Exception Handling)
  - 3.2: Predefined Exception
  - 3.3: Numbered Exceptions
  - 3.4: User Defined Exceptions
  - 3.5: Raising Exceptions
  - 3.6: Control passing to Exception Handlers
  - 3.7: RAISE_APPLICATION_ERROR

Table of Contents

- **Lesson 4: Procedures, Functions, and Packages**
  - 4.1: Subprograms in PL/SQL
  - 4.2: Anonymous Blocks versus Stored Subprograms
  - 4.3: Procedures
  - 4.4: Functions
  - 4.5: Packages
  - 4.6: Autonomous Transactions
- **Lesson 5: Built-in Packages in Oracle**
  - 5.1: Testing and Debugging in PL/SQL
  - 5.2: DBMS_OUTPUT: Displaying Output
  - 5.3: Handling Files (UTL_FILE)
  - 5.4: Handling Large Objects (DBMS_LOB)

Table of Contents

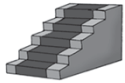
- **Lesson 6: Database Triggers**
  - 6.1: Concept of Database Triggers
  - 6.2: Types of Triggers
  - 6.3: Disabling and Dropping Triggers
  - 6.4: Restriction on Triggers
  - 6.5: Order of Trigger firing
  - 6.6: Using :Old and :New values,
  - 6.7: WHEN clause,
  - 6.8 Examples on Triggers

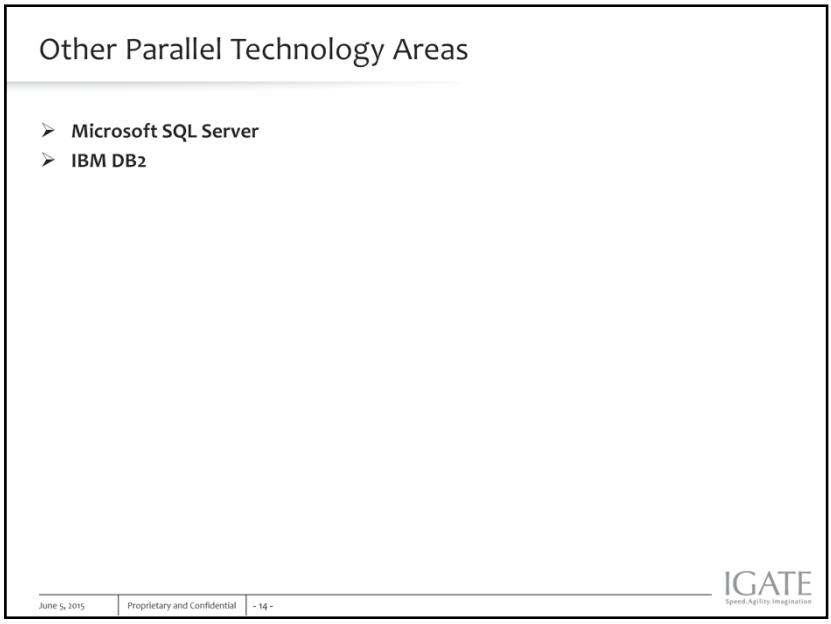
References

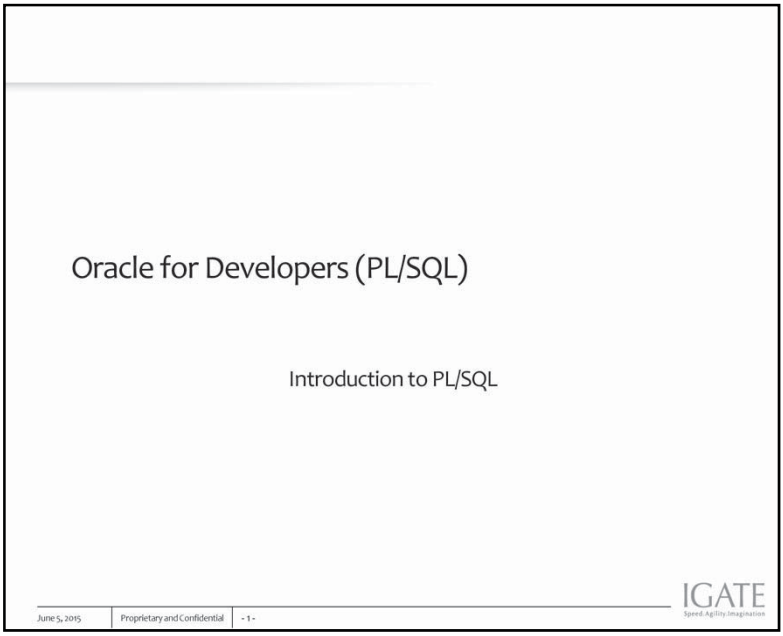
- **Oracle PL/SQL Programming, Third Edition; by Steven Feuerstein**
- **Oracle 9i PL/SQL: A Developer's Guide**
- **Oracle9i PL/SQL Programming; by Scott Urman**

Next Step Courses (if applicable)

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

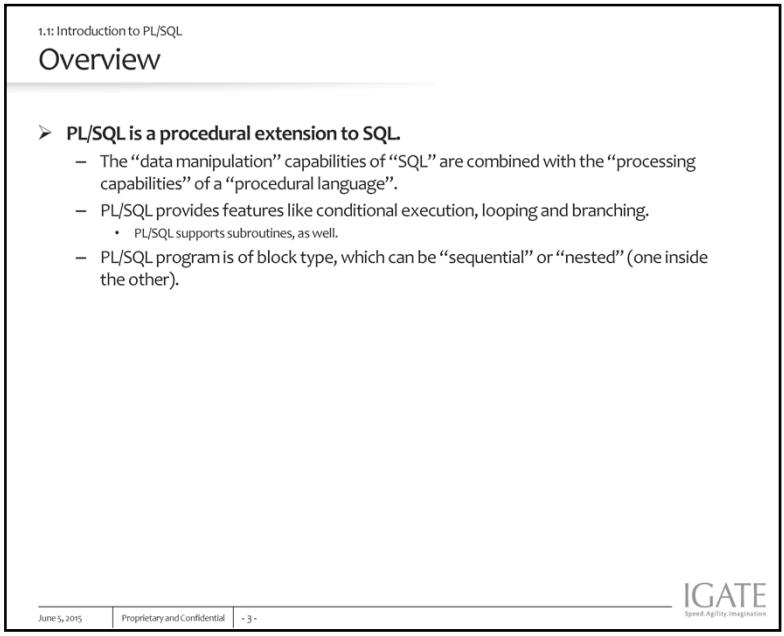



Other Parallel Technology Areas
Microsoft SQL Server
IBM DB2
June 5, 2015
Proprietary and Confidential
- 14 -
IGATE
Speed.Agility.Imagination


Oracle for Developers (PL/SQL)
Introduction to PL/SQL
June 5, 2015
Proprietary and Confidential
- 1 -
IGATE
Speed.Agility.Imagination

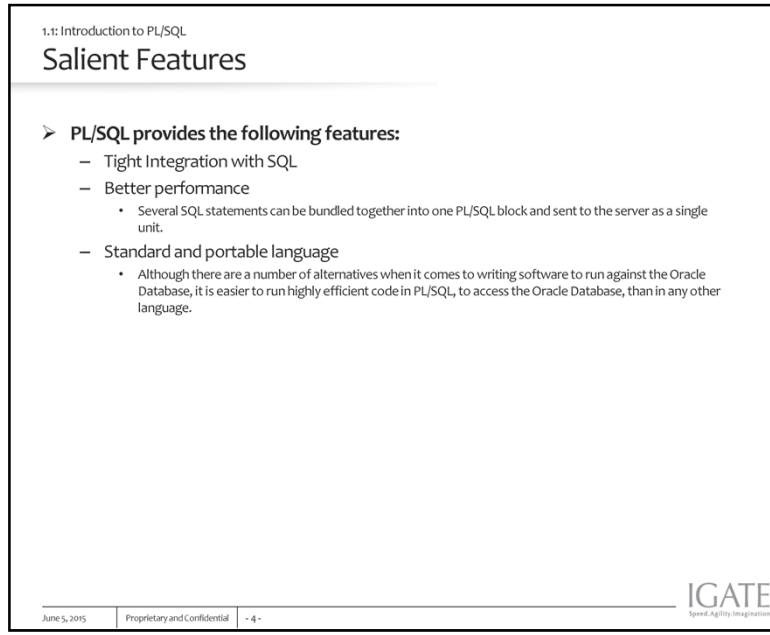
Lesson Objectives

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



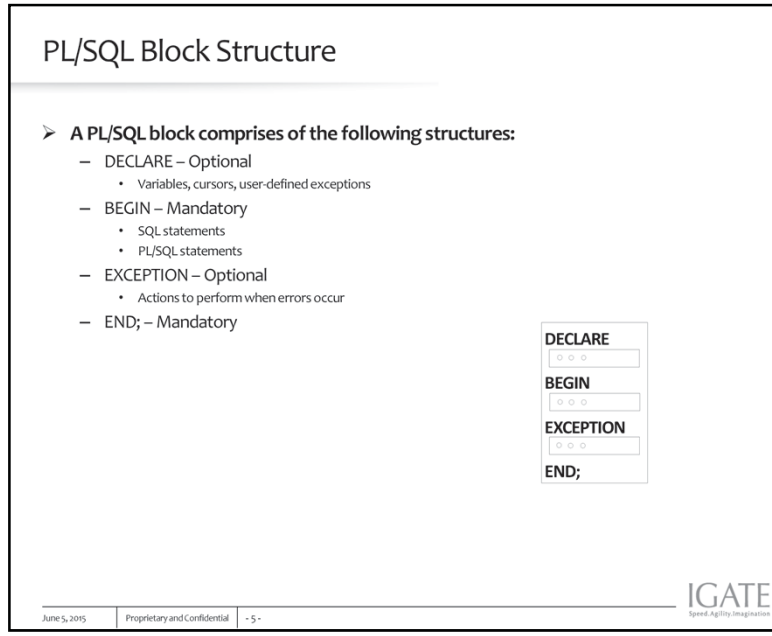
Introduction to PL/SQL:

- 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.



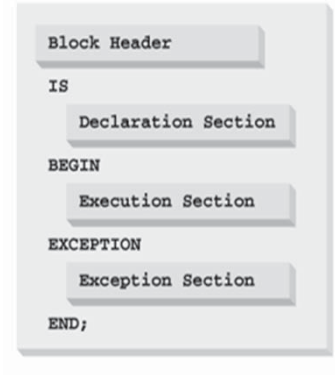
PL/SQL Block Structure:

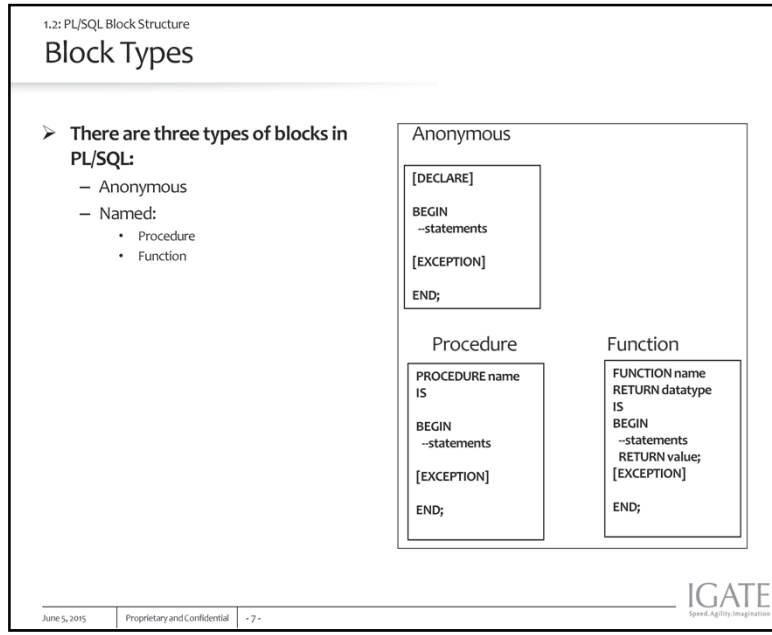
- PL/SQL is a block-structured language. Each basic programming unit that is written to build your application is (or should be) a "logical unit of work". The PL/SQL block allows you to reflect that logical structure in the physical design of your programs.
- Each PL/SQL block has up to four different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





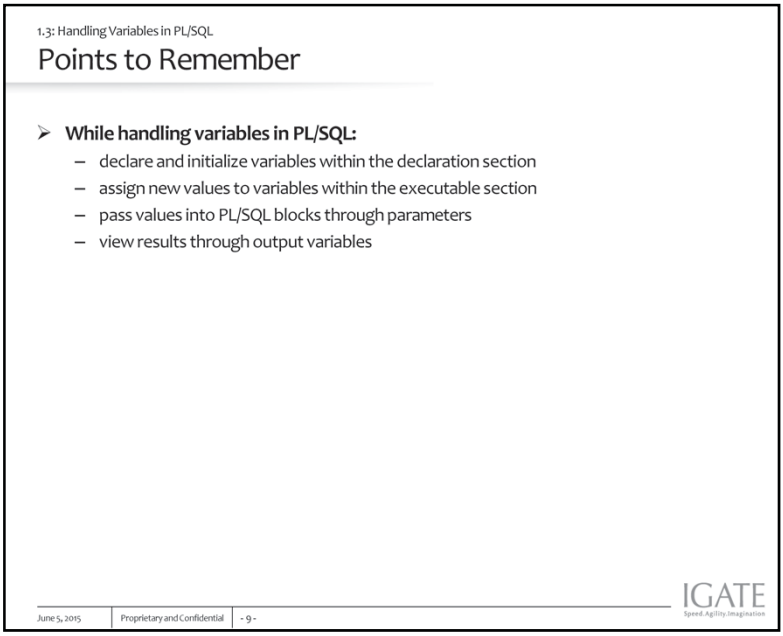
Block Types:

- 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:

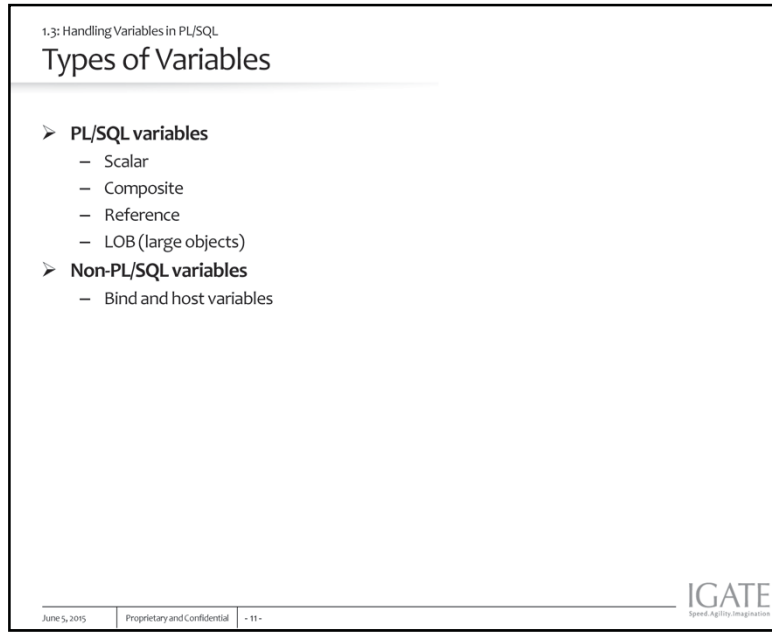
- The notations used in a PL/SQL block are given below:
  1. -- is a single line comment.
  2. /* */ is a multi-line comment.
  3. Every statement must be terminated by a semicolon (;).
  4. PL/SQL block is terminated by a slash (/) on a line by itself.
- A PL/SQL block must have an "Execution section".
- 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'
        FOR UPDATE of sal ;
        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
June 5, 2015
Proprietary and Confidential
- 9 -
IGATE
Speed.Agility.Imagination

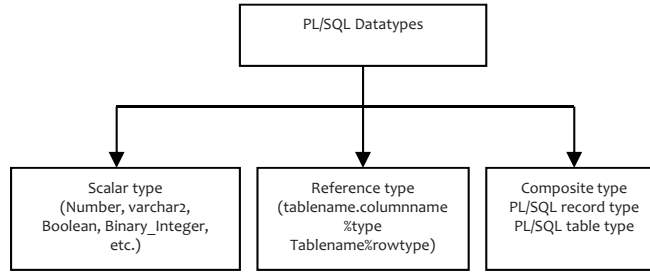
Guidelines for declaring variables

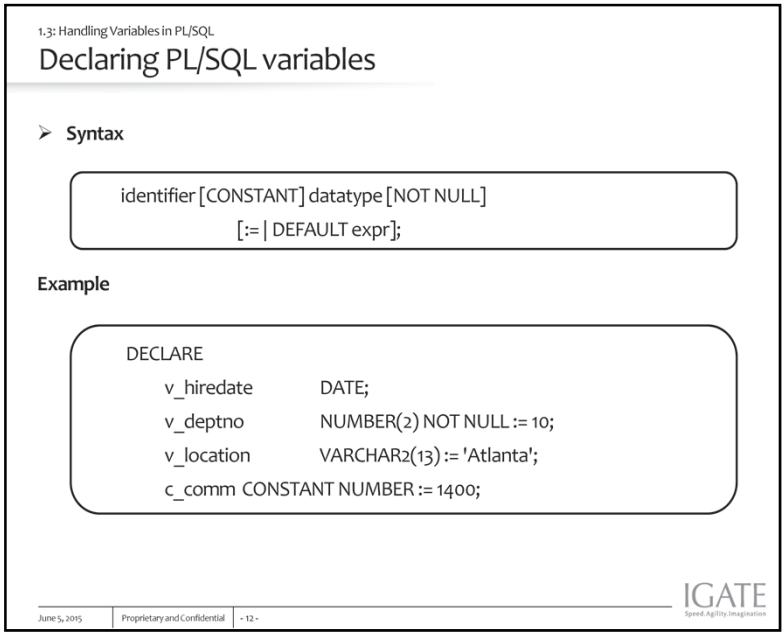
- **Given below are a few guidelines for declaring variables:**
  - follow the naming conventions
  - initialize the variables designated as NOT NULL
  - initialize the identifiers by using the assignment operator (:=) or by using the DEFAULT reserved word
  - declare at most one Identifier per line



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];
```

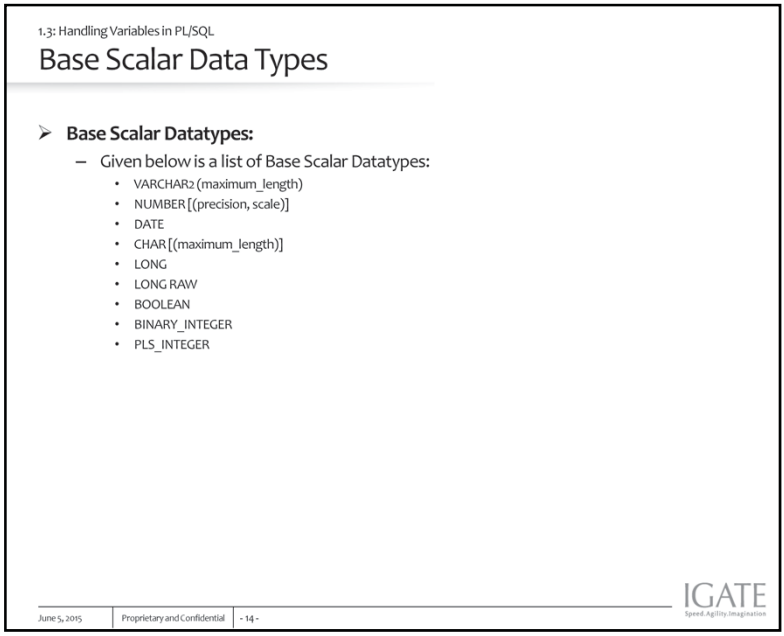
  - In the syntax given above:
    - **identifier** is the name of the variable.
    - **CONSTANT** constrains the variable so that its value cannot change. Constants must be initialized.
    - **datatype** is a scalar, composite, reference, or LOB datatype.
    - **NOT NULL** constrains the variable so 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:

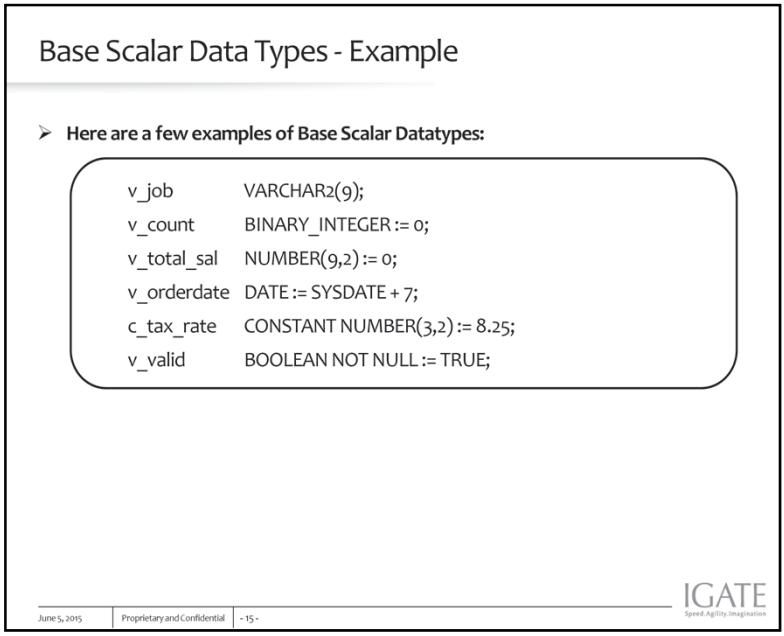
```
DECLARE
   v_description          varchar2 (25);
   v_sal                  number (5) not null: = 3000;
   v_compcode             varchar2 (20) constant: =
'abc
                      consultants';
   v_comm       not null default   0;
```



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.

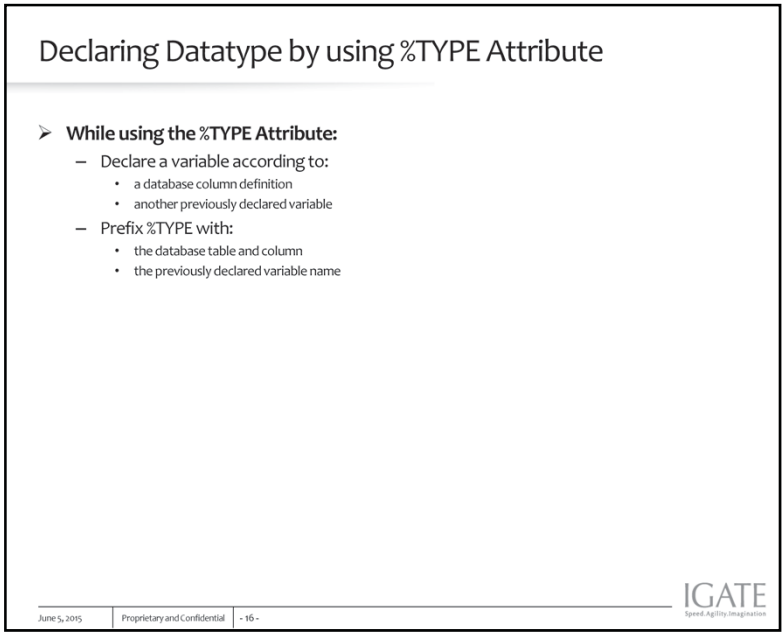
contd.



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('Diamet
er: '||2*pie*radius);
end;
/
```



Reference types:

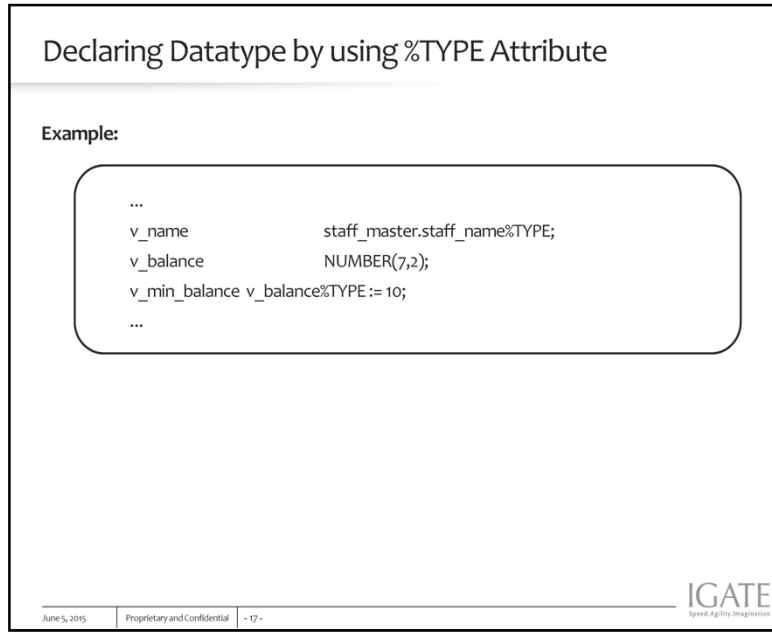
- A "reference type" in PL/SQL is the same as a "pointer" in C. A "reference type" variable can point to different storage locations over the life of the program.

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;
```

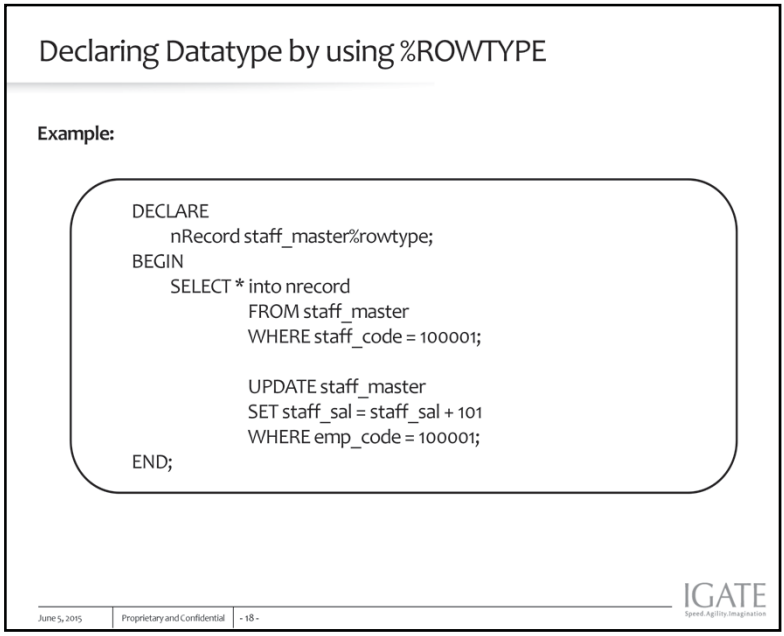
  - **Note:** Datatype of V_Empno is same as datatype of Empno column of the EMP table.



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;
/
```

Inserting and Updating using records

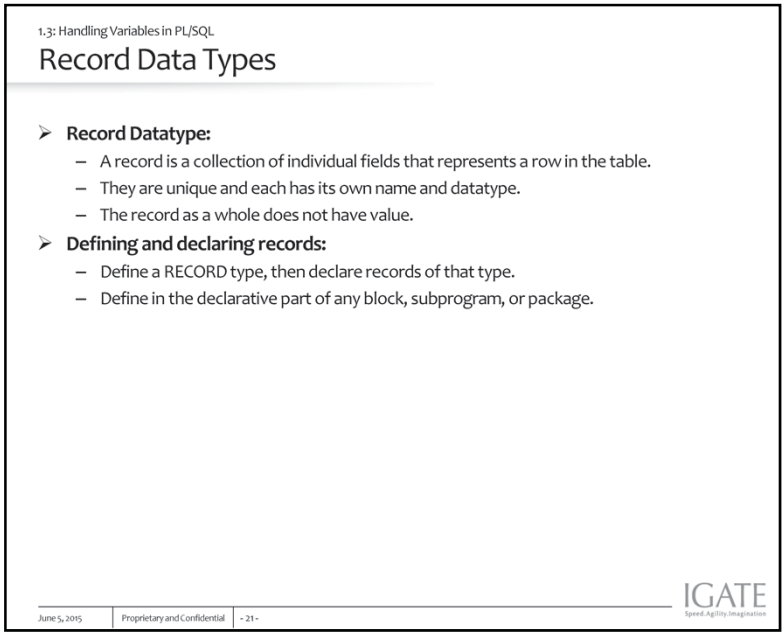
Example:

```
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;
```

1.3: Handling Variables in PL/SQL

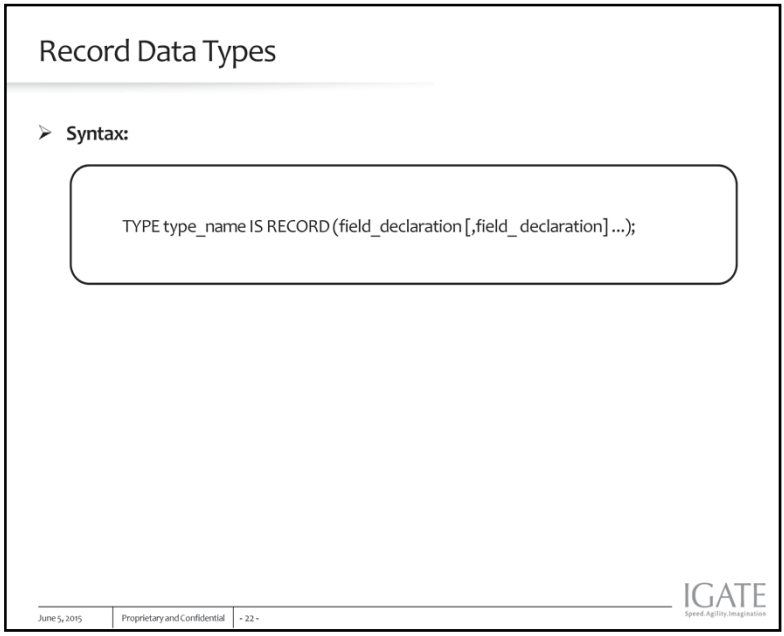
Composite Data Types

- **Composite Datatypes in PL/SQL:**
  - Two composite datatypes are available in PL/SQL:
    - records
    - tables
  - A composite type contains components within it. A variable of a composite type contains one or more scalar variables.



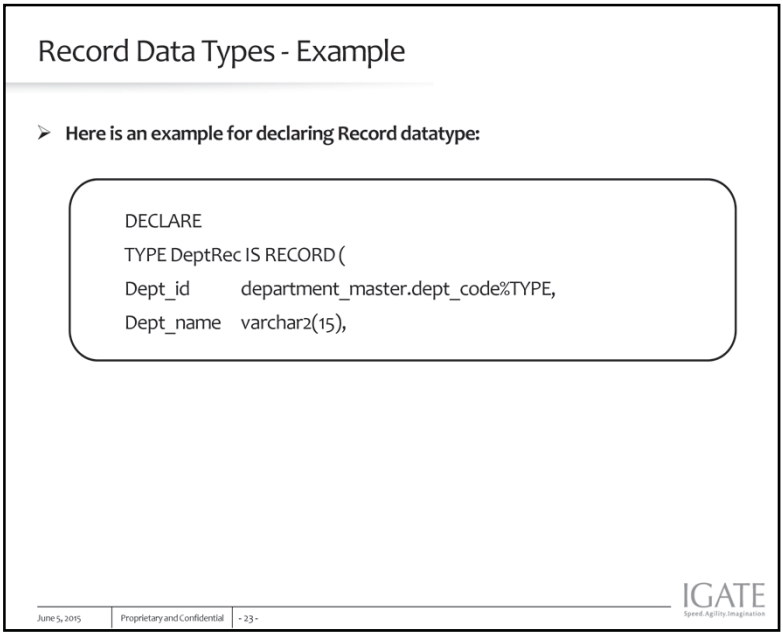
Record Datatype:

- 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);

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;
```

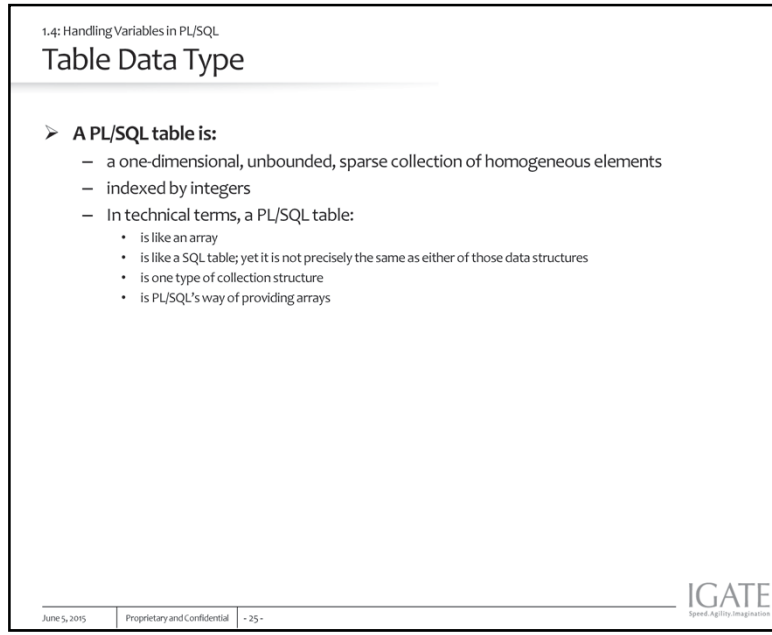
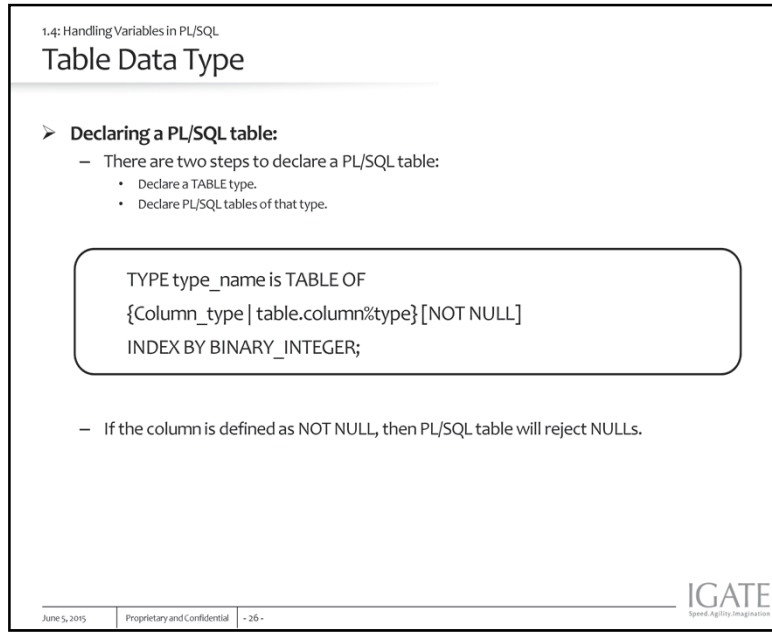


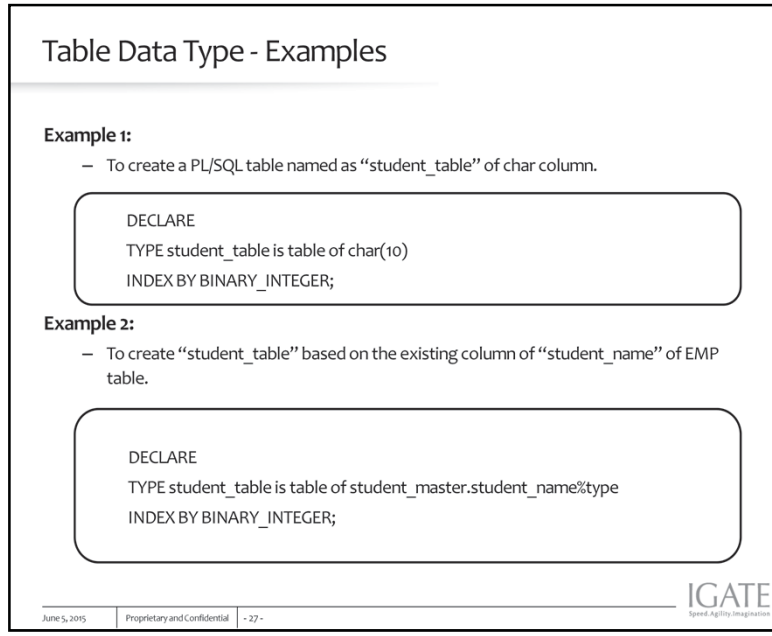
Table Datatype

- Like PL/SQL records, the table is another composite datatype. PL/SQL tables are objects of type TABLE, and look similar to database tables but with slight difference.
- PL/SQL tables use a primary key to give you array-like access to rows.
  - Like the size of the database table, the size of a PL/SQL table is unconstrained. That is, the number of rows in a PL/SQL table can dynamically increase. So your PL/QSL table grows as new rows are added.
  - PL/SQL table can have one column and a primary key, neither of which can be named.
    - The column can have any datatype, but the primary key must be of the type BINARY_INTEGER.
- Arrays are like temporary tables in memory. Thus they are processed very quickly.
- Like the size of the database table, the size of a PL/SQL table is unconstrained.
- The "column" can have any datatype. However, the "primary key" must be of the type BINARY_INTEGER.



Declaring a PL/SQL table

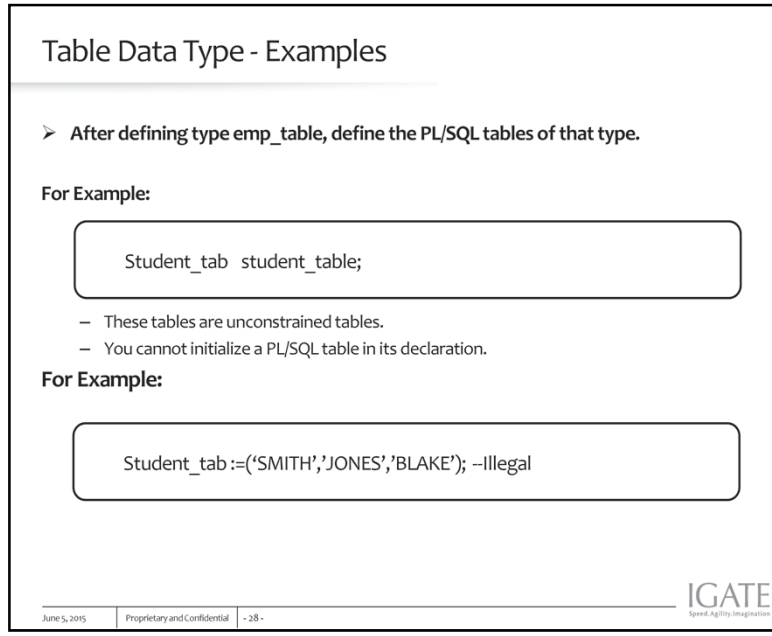
- PL/SQL tables must be declared in two steps. First you declare a TABLE type, then declare PL/SQL tables of that type. You can declare TABLE type in the declarative part of any block, subprogram or package.
- In the syntax on the above slide:
  - Type_name is type specifier used in subsequent declarations to define PL/SQL tables and column_name is any datatype.
  - You can use %TYPE attribute to specify a column datatype. If the column to which table.column refers is defined as NOT NULL, the PL/SQL table will reject NULLs.



Declaring a PL/SQL table (contd.):

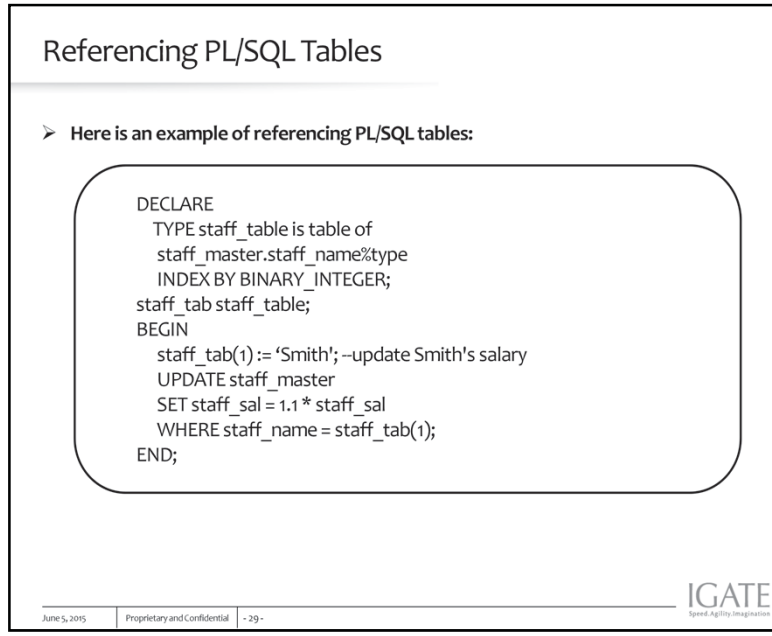
- Example 3:
  - To declare a NOT NULL constraint
  - **Note:** INDEX BY BINARY INTERGER is a mandatory feature of the PL/SQL table declaration.

```
DECLARE
TYPE student_table is table of
student_master.student_name%TYPE NOT NULL
INDEX BY BINARY_INTEGER;
```



Note:

- The PL/SQL tables are unconstrained tables, because its primary key can assume any value in the range of values defined by BINARY_INTEGER.
- You cannot initialize a PL/SQL table in its declaration.

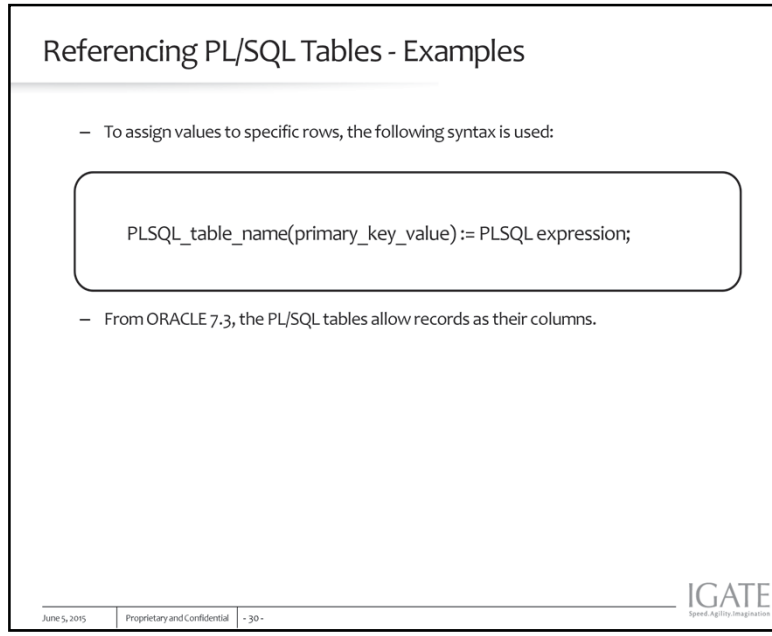


Referencing PL/SQL tables:

- To reference rows in a PL/SQL table, you specify the PRIMARY KEY value using the array-like syntax as shown below:

```
PL/SQL table_name (primary key value)
```

- When primary key value belongs to type BINARY_INTEGER you can reference the first row in PL/SQL table named emp_tab as shown in the slide.

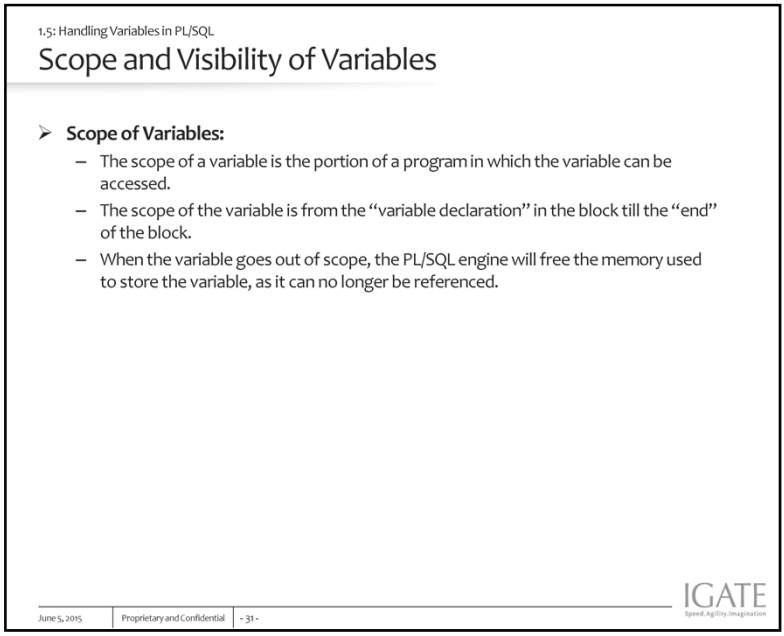


Referencing PL/SQL tables:
Examples:

- Referencing fields of record elements in PL SQL tables:

```
type staff_rectype is record (
staff_id integer,
staff_sname varchar2(60)) ;

type staff_table is table of staff_rectype
index by binary_integer;

staff_tab          staff_table;
```

```
staff_tab(375).staff_sname := 'SMITH';
```



Scope and Visibility of Variables:

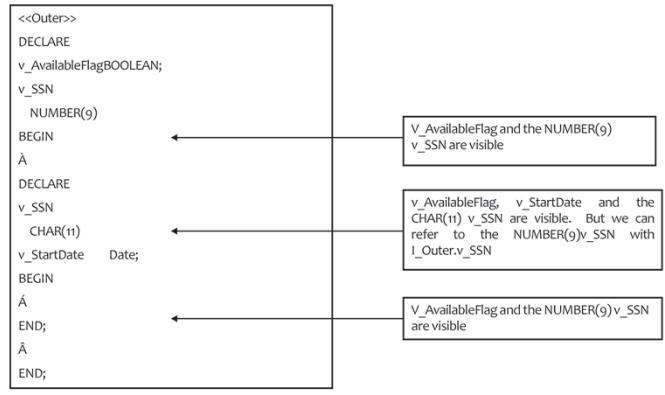
- References to an identifier are resolved according to its scope and visibility.
  - The scope of an identifier is that region of a program unit (block, subprogram, or package) from which you can reference the identifier.
  - An identifier is visible only in the regions from which you can reference 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.

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.

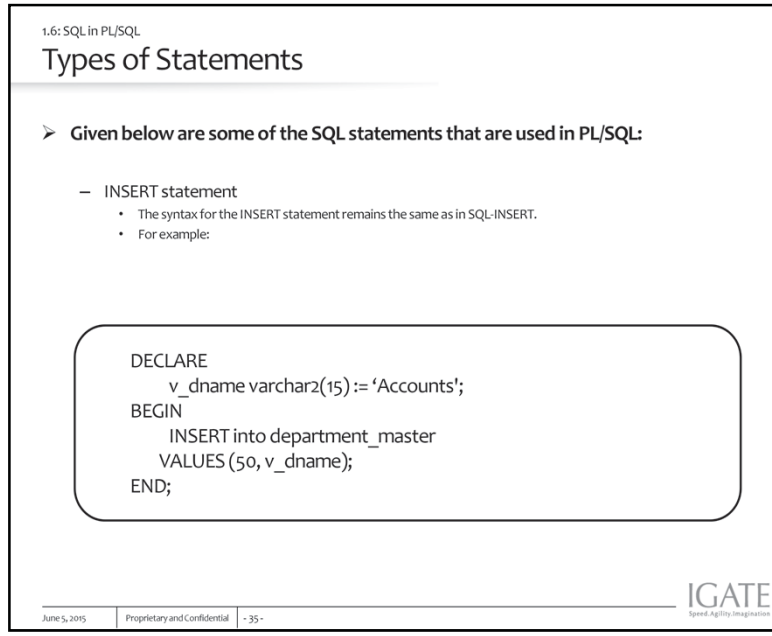
Scope and Visibility of Variables

- **Pictorial representation of visibility of a variable:**



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;
```


1.6: 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;
June 5, 2015
Proprietary and Confidential
- 35 -
IGATE
Speed.Agility.Imagination

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;
```

Types of Statements

- UPDATE statement

For Example:

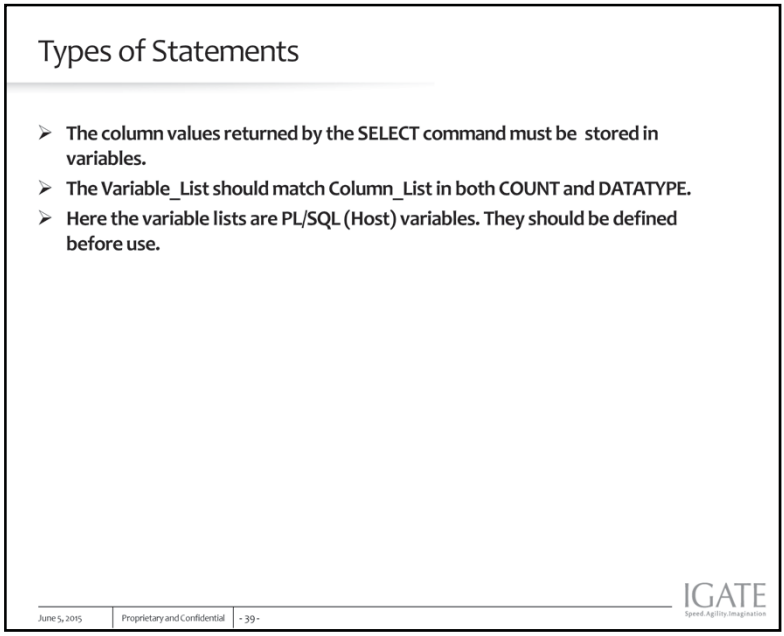
```
DECLARE
      v_sal_incr  number(5) := 1000;
BEGIN
      UPDATE staff_master
    SET staff_sal = staff_sal + v_sal_incr
    WHERE staff_name='Smith';
END;
```


.

Types of Statements

- SELECT statement
  - Syntax:

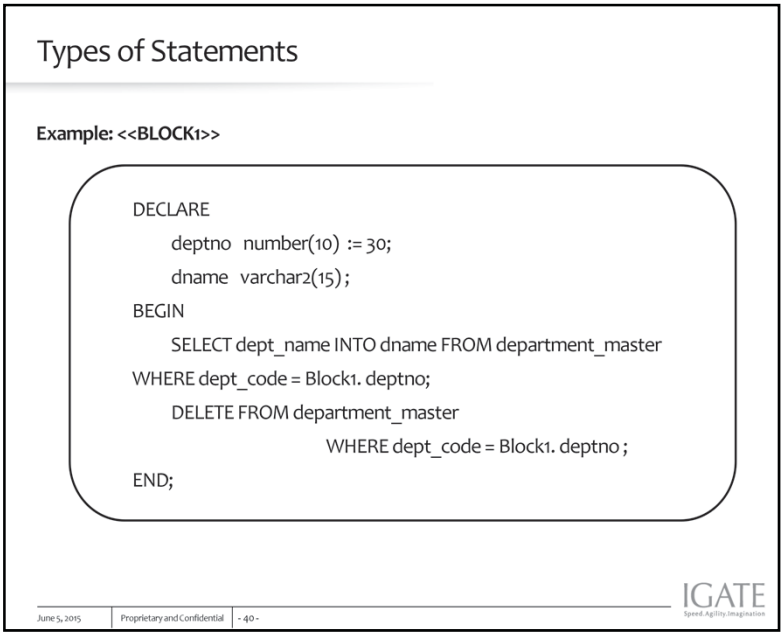
```
SELECT Column_List INTO Variable_List
    FROM Table_List
    [WHERE expr1]
    CONNECT BY expr2 [START WITH expr3]]
    GROUP BY expr4] [HAVING expr5]
    [UNION | INTERSECT | MINUS SELECT ...]
    [ORDER BY expr | ASC | DESC]
    [FOR UPDATE [OF Col1,...] [NOWAIT]]
    INTO Variable_List;
```



SELECT Statement:

Note:

- The SELECT clause is used if the selected row must be modified through a DELETE or UPDATE command.
- Since the contents of the row must not be modified between the SELECT command and the UPDATE command, it is necessary to lock the row after the SELECT command.
  - FOR UPDATE will lock the selected row.
  - OF Col1,.. lists the columns which can be modified by the UPDATE command.
    - If OF Col1,... is missing, all the columns can be modified.
- It is possible that when SELECT..UPDATE is issued, the concerned row is already locked by some other user or a previous SELECT..UPDATE command.
  - If NOWAIT is not given, then the current SELECT..UPDATE command will wait until the lock is cleared.
  - IF NOWAIT is given, then the SELECT..UPDATE will return immediately with a failure.



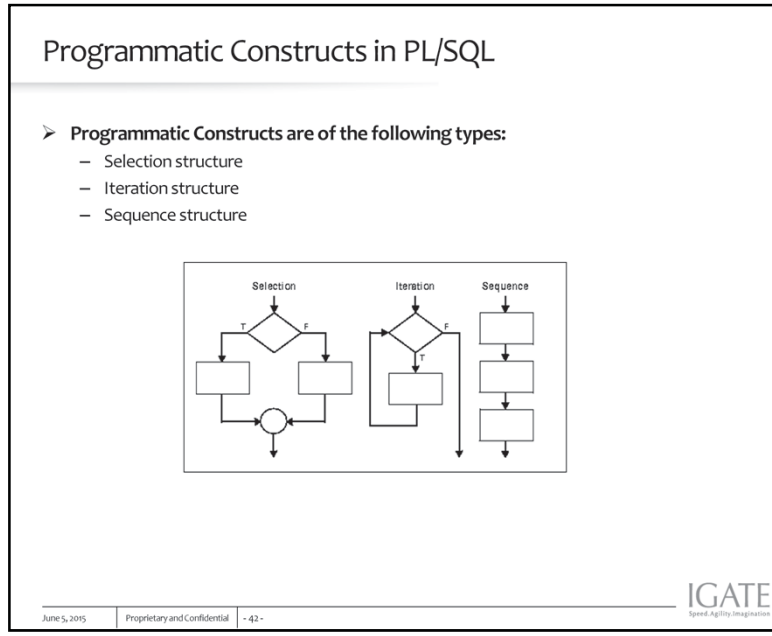
SELECT statement (contd.):

SELECT statement (contd.):

More examples

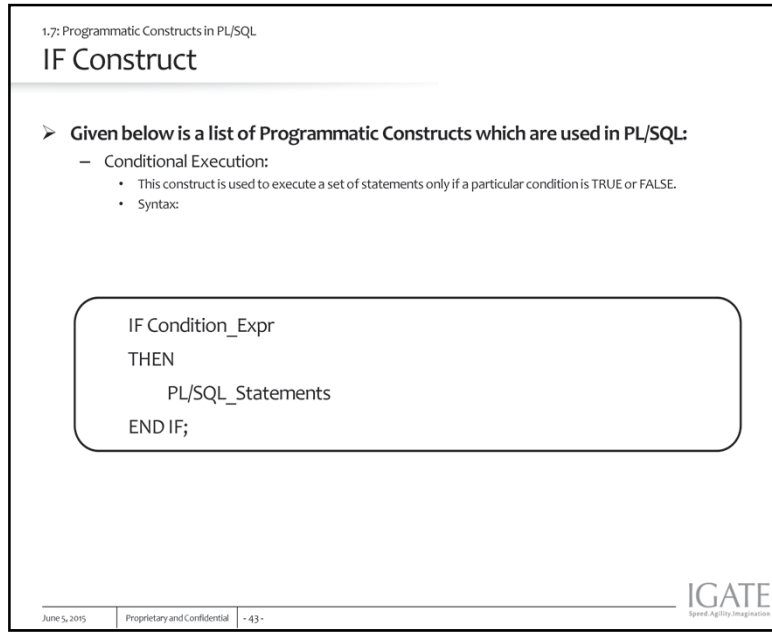
- Here the SELECT statement will select names of all the departments and not only deptno. 30.
- The DELETE statement will delete all the employees.
  - This happens because when the PL/SQL engine sees a condition expr1 = expr2, the expr1 and expr2 are first checked to see whether they match the database 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.

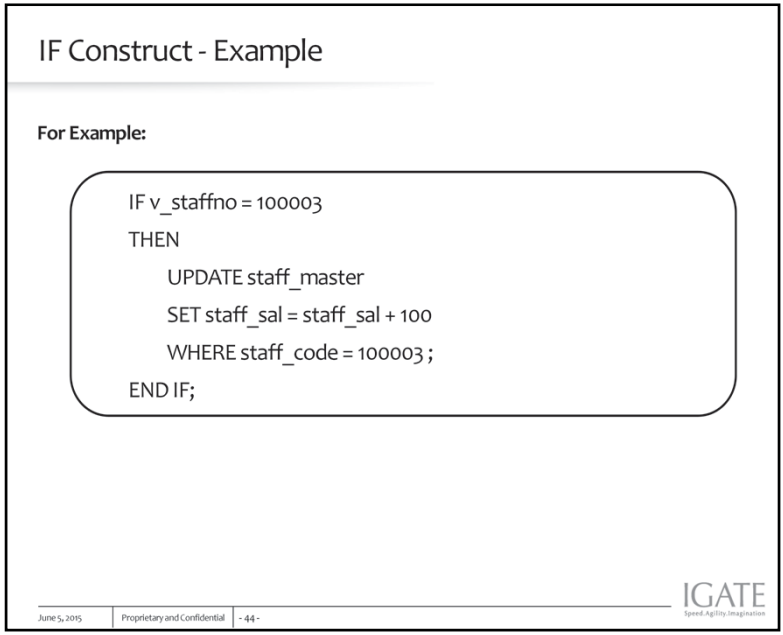


Programmatic Constructs (contd.)

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.

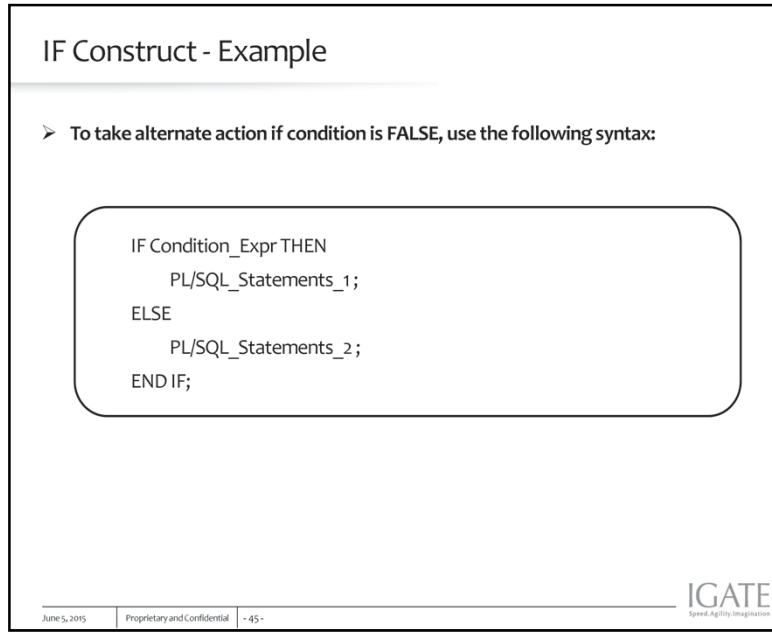
contd.



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.

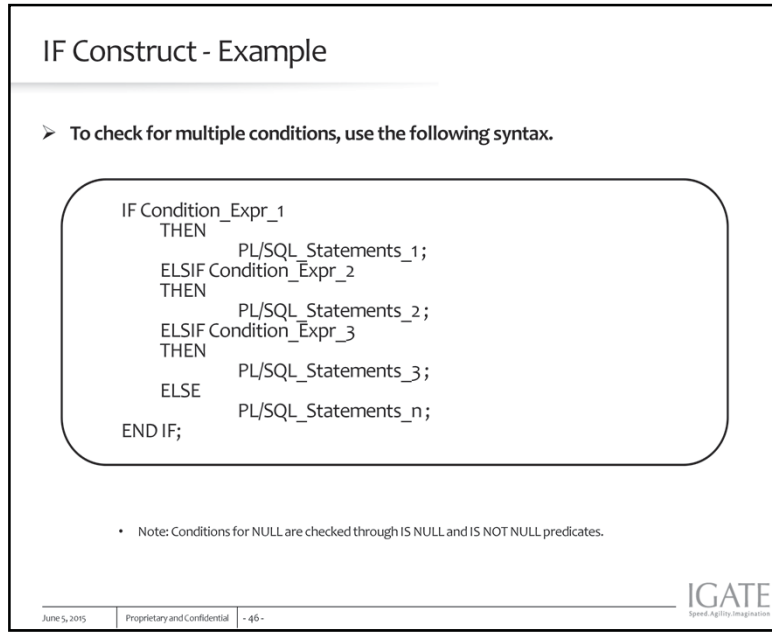


Programmatic Constructs (contd.)

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.):

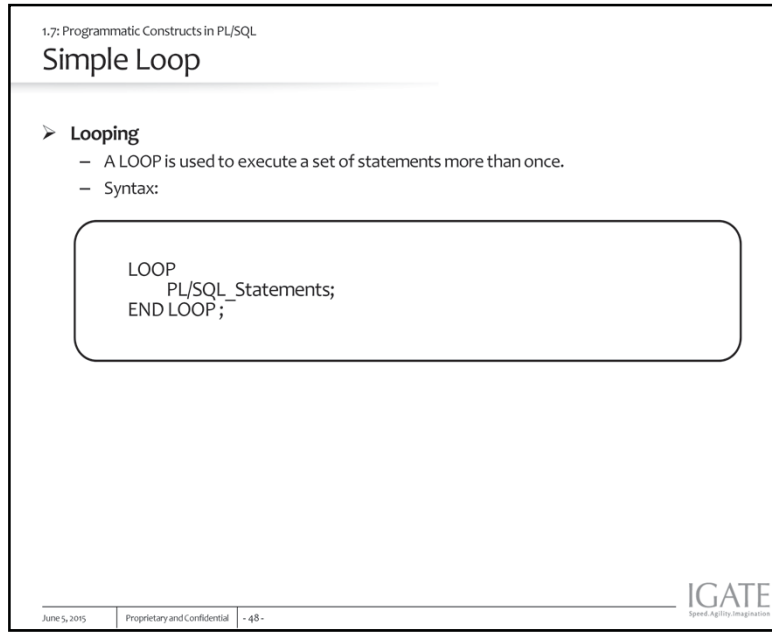
```
DECLARE
    D VARCHAR2(3): = TO_CHAR(SYSDATE, 'DY')
BEGIN
    IF   D= 'SAT' THEN
        DBMS_OUTPUT.PUT_LINE('ENJOY YOUR
WEEKEND');
    ELSIF    D= 'SUN'     THEN
        DBMS_OUTPUT.PUT_LINE('ENJOY YOUR
WEEKEND');
    ELSE
        DBMS_OUTPUT.PUT_LINE('HAVE A NICE
DAY');
    END IF;
END;
```

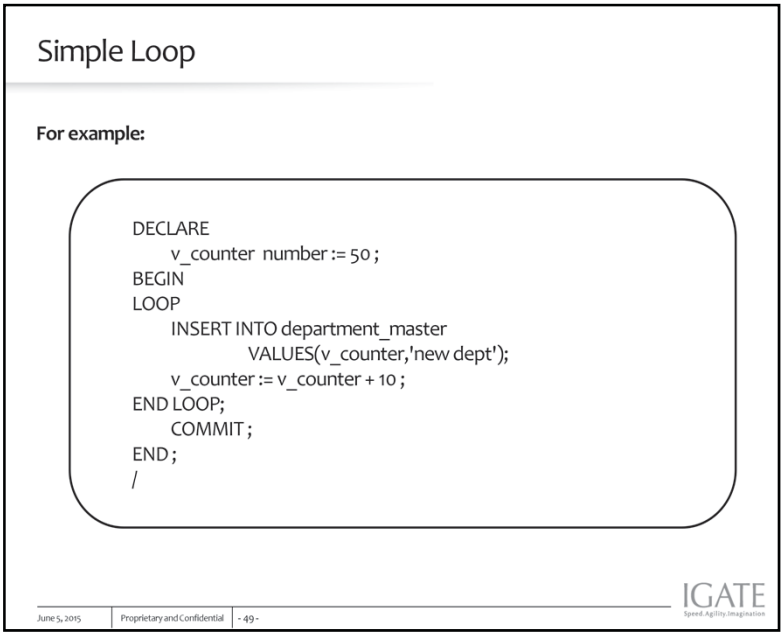
Programmatic Constructs (contd.)

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;
```


1.7: 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 ;
June 5, 2015
Proprietary and Confidential
- 48 -
IGATE
Speed.Agility.Imagination



Programmatic Constructs (contd.)

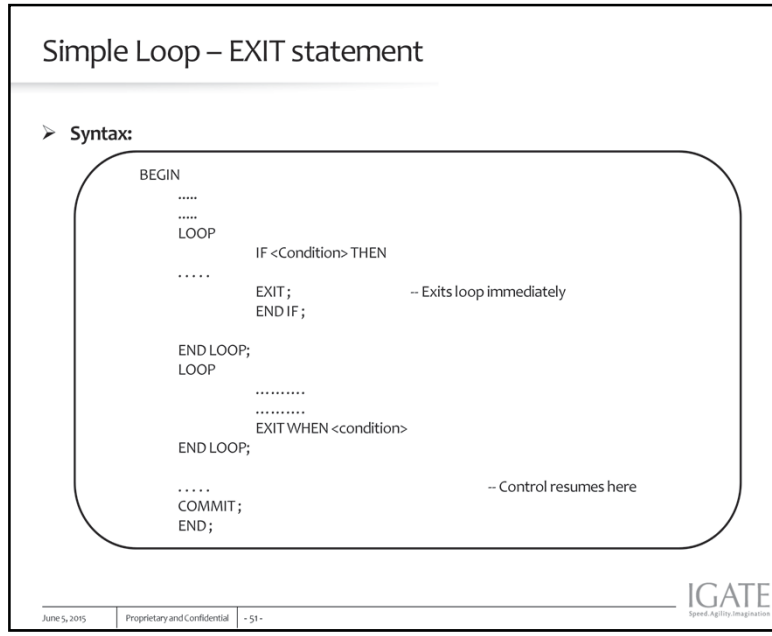
Looping

- 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.

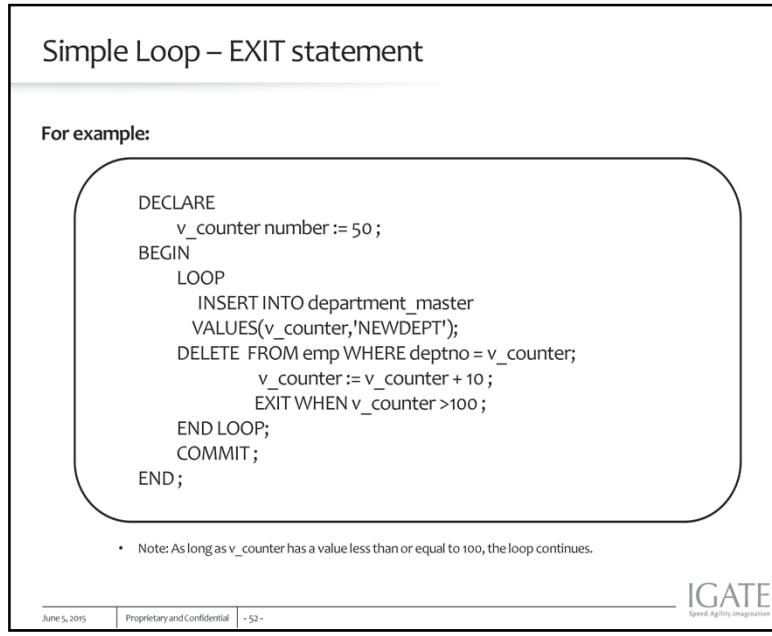
Simple Loop – EXIT statement

- **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.

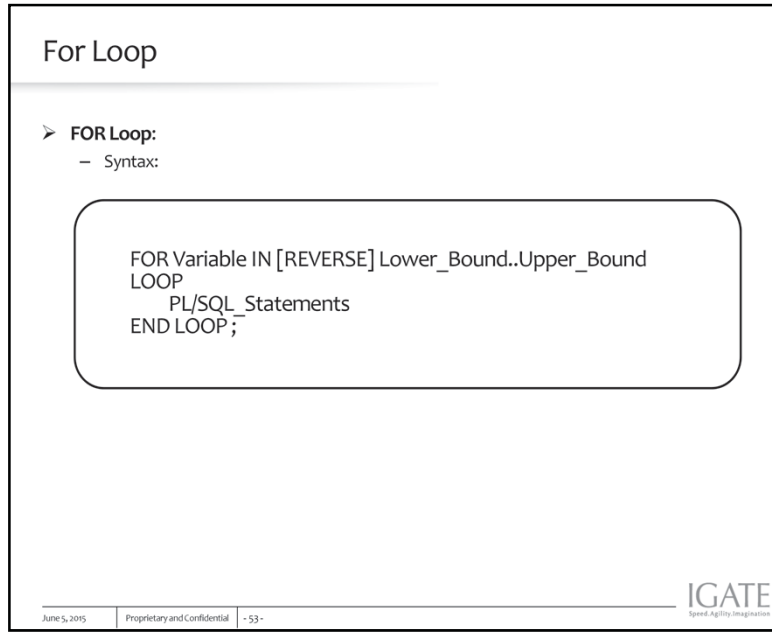
contd.



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



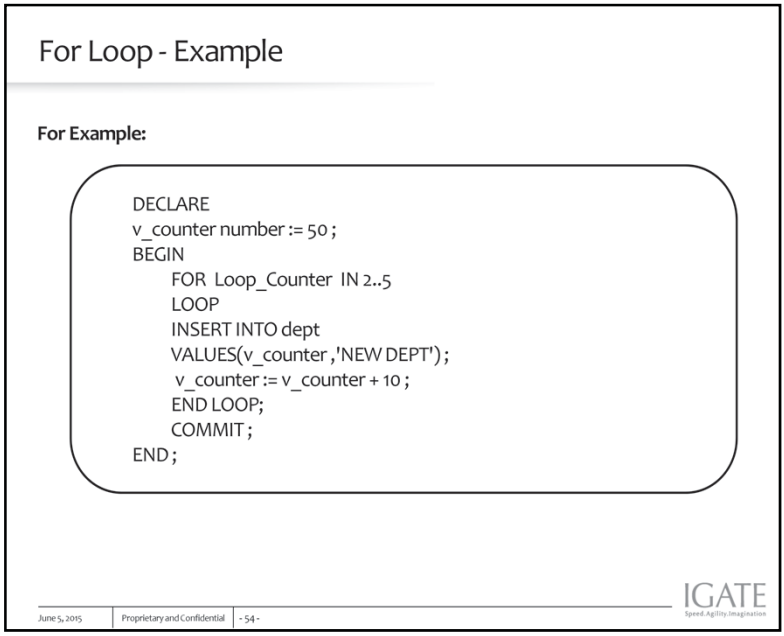
Note:
LOOP.. END LOOP can be used in conjunction with FOR and WHILE for better control on looping.



Programmatic Constructs (contd.)

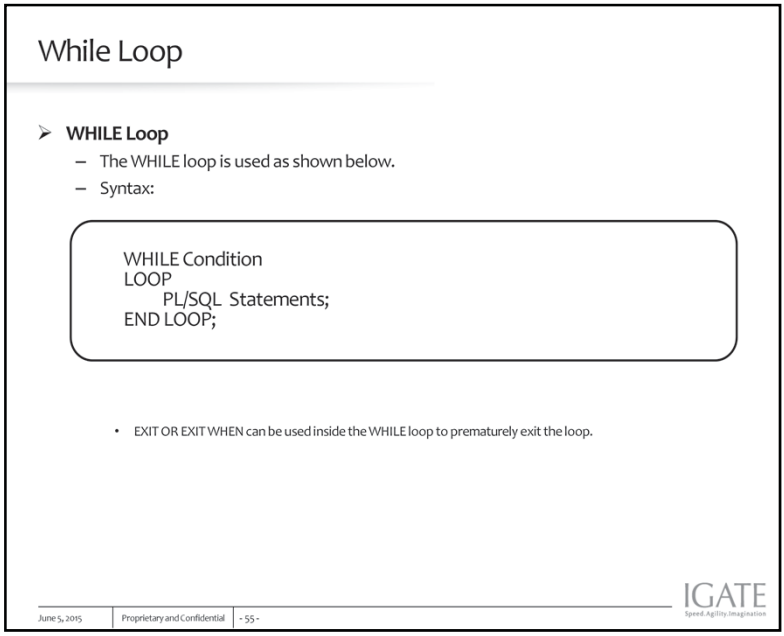
FOR Loop:

- 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.

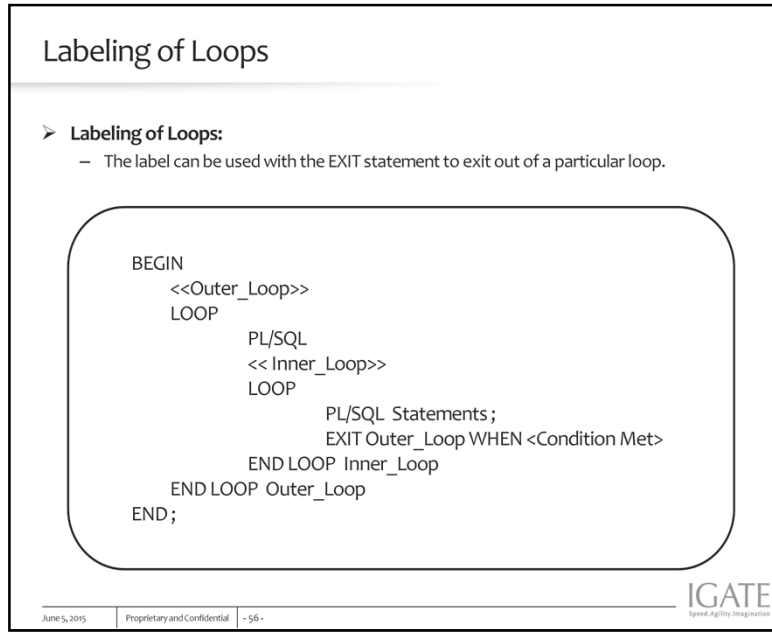


Programmatic Constructs (contd.)

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
    - pass values into PL/SQL blocks through parameters
    - view results through output variables

Summary

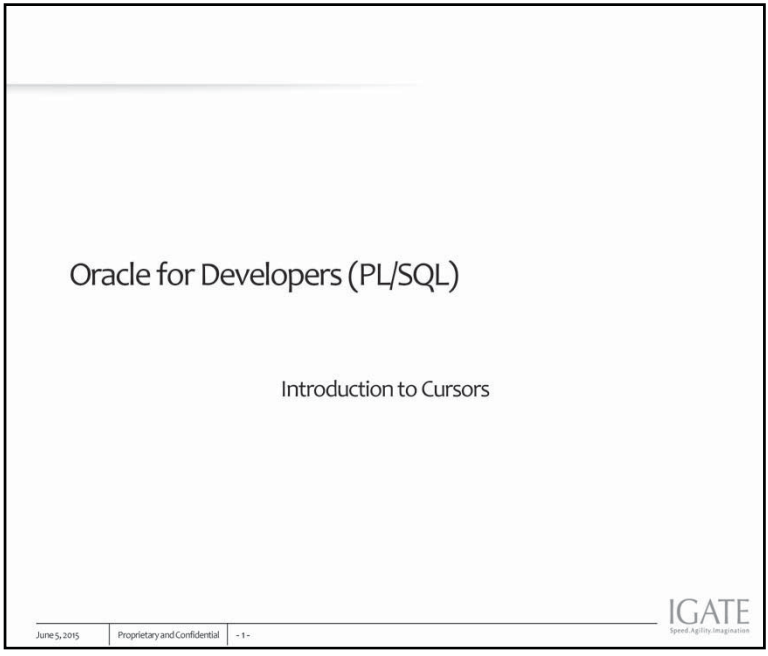
- Different types of PL/SQL Variables are: Scalar, Composite, Reference, LOB
- Scope of a variable: It is the portion of a program in which the variable can be accessed.
- Visibility of a variable: It is the portion of the program, where the variable can be accessed without having to qualify the reference.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

Review – Questions

- 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: PL/SQL tables use a primary key to give you array-like access to rows.**
  - True / False

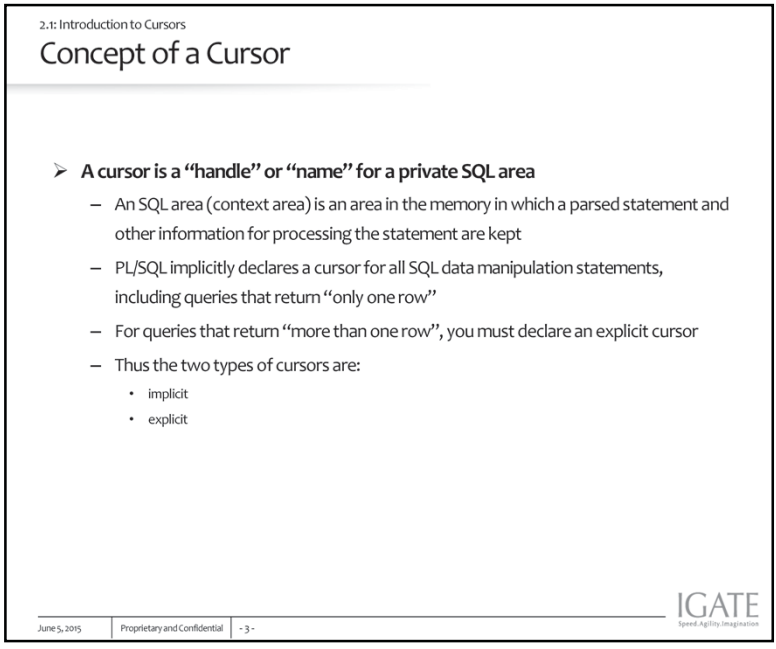
Review – Questions

- Question 4: **While using FOR loop, Upper_Bound, and Lower_Bound must be integers.**
  - True / False


Oracle for Developers (PL/SQL)
Introduction to Cursors
June 5, 2015
Proprietary and Confidential
- 1 -
IGATE
Speed. Agility. Imagination

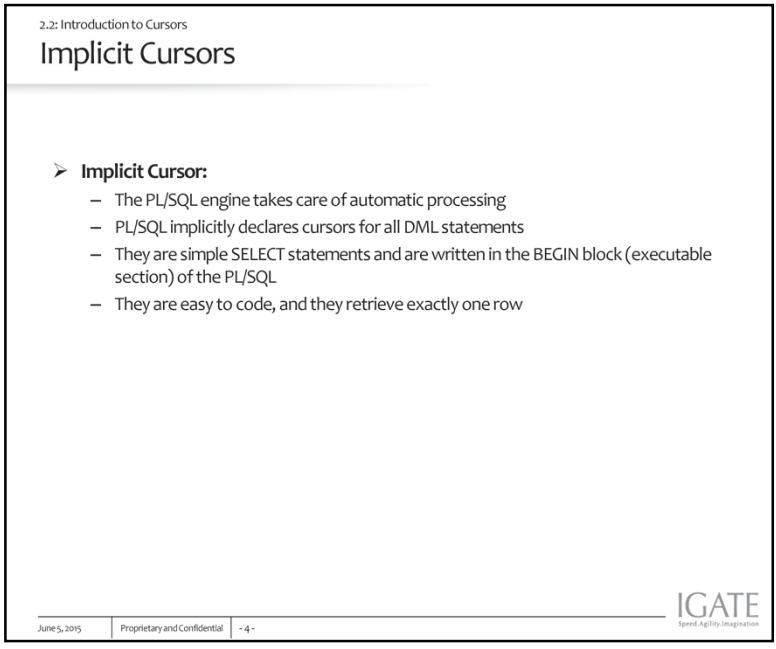
Lesson Objectives

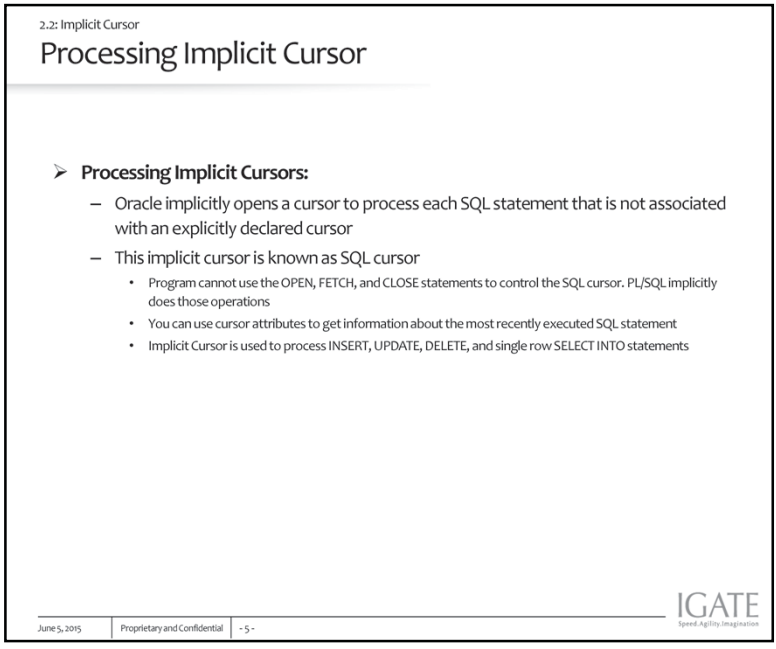
- **To understand the following topics:**
  - Introduction to Cursors
  - Implicit Cursors
  - Explicit Cursors
  - Cursor attributes
  - Processing Implicit Cursors and Explicit Cursors
  - Cursor with Parameters
  - Use of Cursor Variables
  - Difference between Cursors and Cursor Variables



Introduction to Cursors:

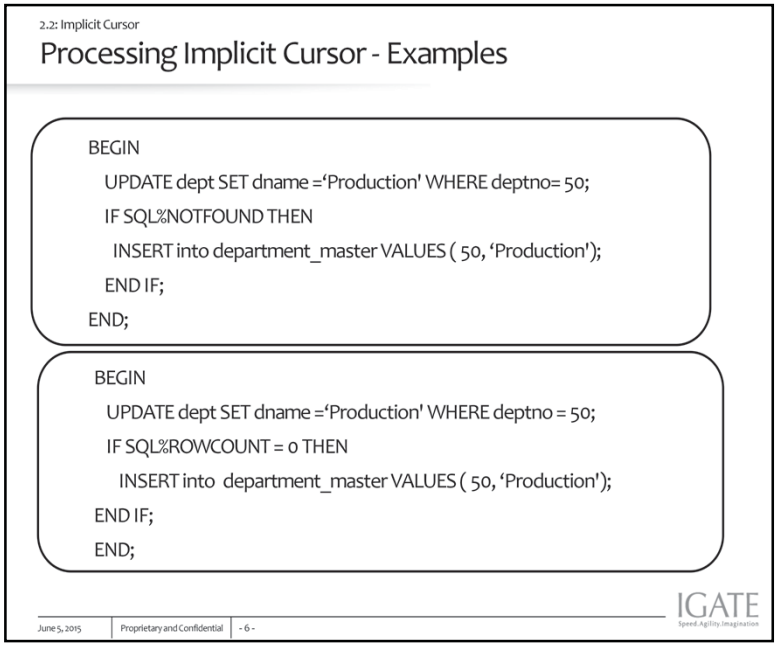
- ORACLE allocates memory on the Oracle server to process SQL statements. It is called as "**context area**". Context area stores information like number of rows processed, the set of rows returned by a query, etc.
- A Cursor is a "handle" or "pointer" to the context area. Using this cursor the PL/SQL 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: Introduction to Cursors
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
June 5, 2015
Proprietary and Confidential
- 4 -
IGATE
Speed.Agility.Imagination



Processing Implicit Cursors:

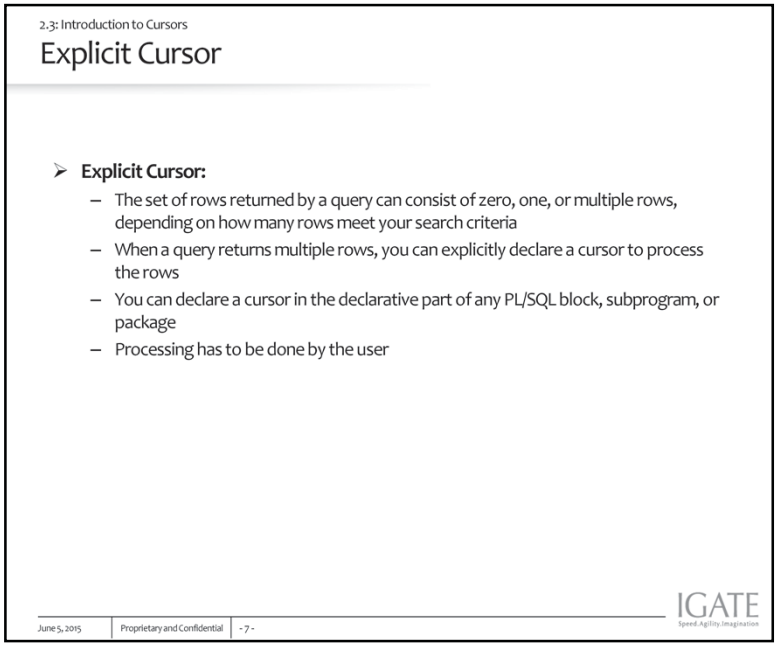
- All SQL statements are executed inside a context area and have a cursor, which points to that context area. This implicit cursor is known as SQL cursor.
- Implicit SQL cursor is not opened or closed by the program. PL/SQL implicitly opens the cursor, processes the SQL statement, and closes the cursor.
- 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:

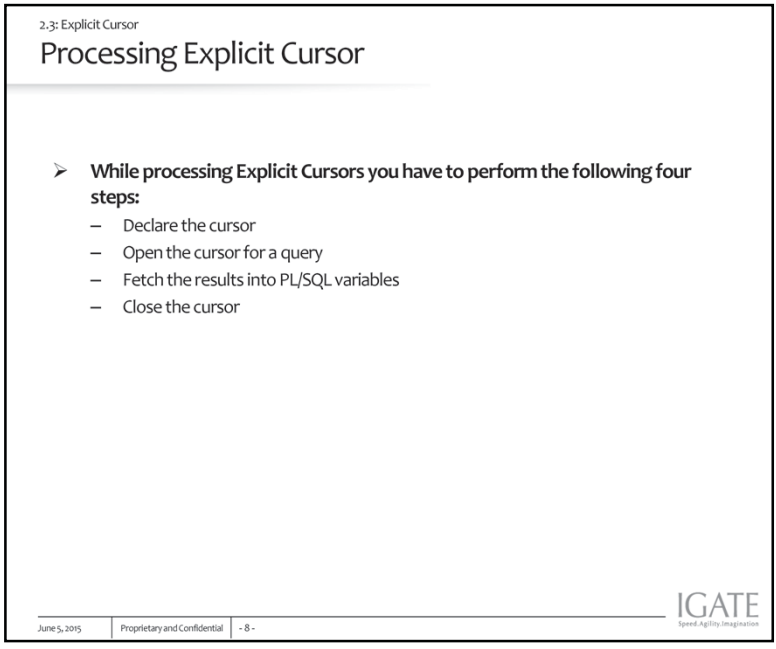
Note:

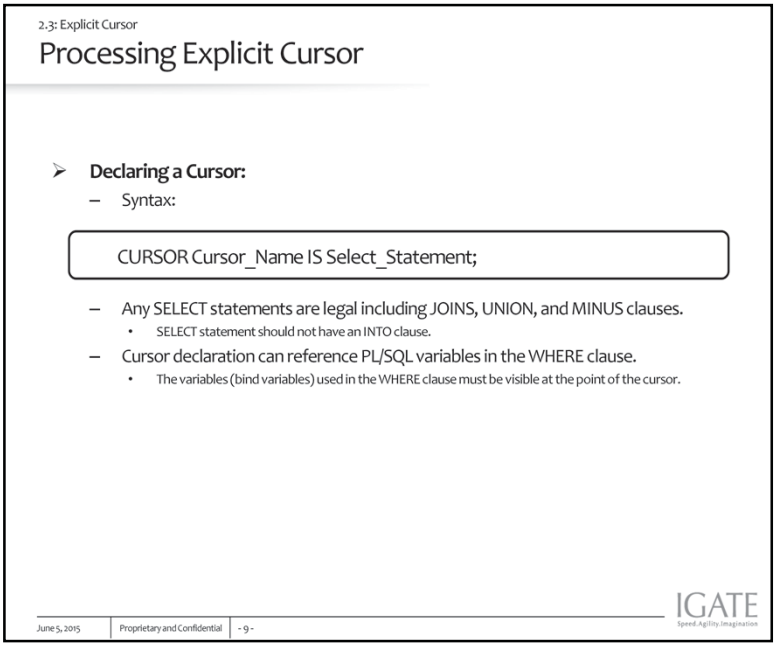
- SQL%NOTFOUND should not be used with SELECT INTO …. Statement.
- This is because SELECT INTO…. Statement will raise an ORACLE error if no rows are selected, and
  - control will pass to exception * section (discussed later), and
  - SQL%NOTFOUND statement will not be executed at all
- The slide shows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

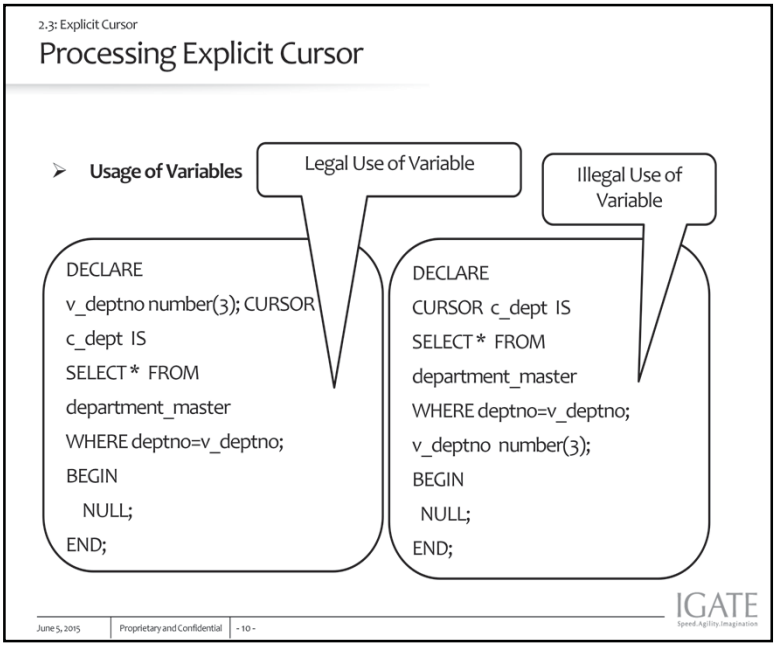


Explicit Cursor:

- 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.

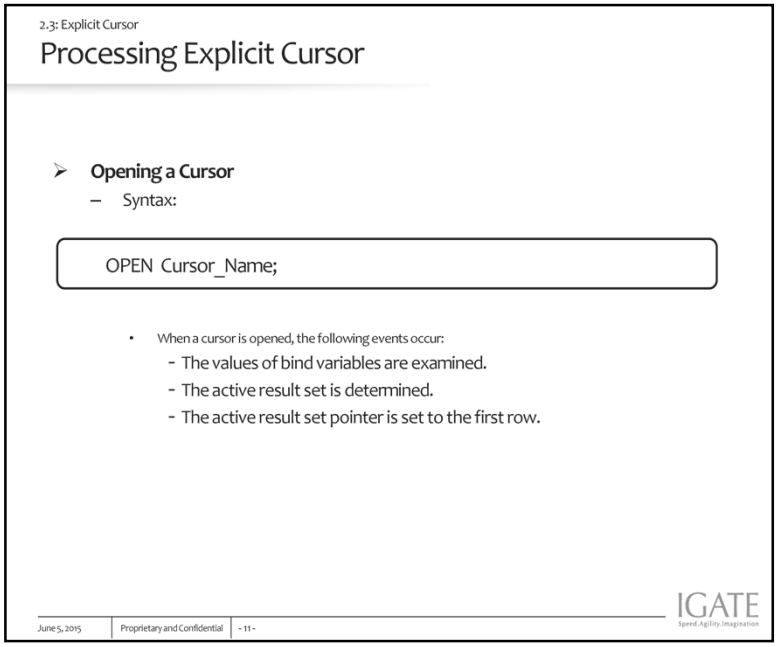

2.3: Explicit Cursor
Processing Explicit Cursor
While processing Explicit Cursors you have to perform the following four steps:
Declare the cursor
Open the cursor for a query
Fetch the results into PL/SQL variables
Close the cursor
June 5, 2015
Proprietary and Confidential
- 8 -
IGATE
Speed.Agility.Imagination


2.3: Explicit Cursor
Processing Explicit Cursor
Declaring a Cursor:
Syntax:
CURSOR Cursor_Name IS Select_Statement;
Any SELECT statements are legal including JOINS, UNION, and MINUS clauses.
SELECT statement should not have an INTO clause.
Cursor declaration can reference PL/SQL variables in the WHERE clause.
The variables (bind variables) used in the WHERE clause must be visible at the point of the cursor.
June 5, 2015
Proprietary and Confidential
- 9 -
IGATE
Speed.Agility.Imagination



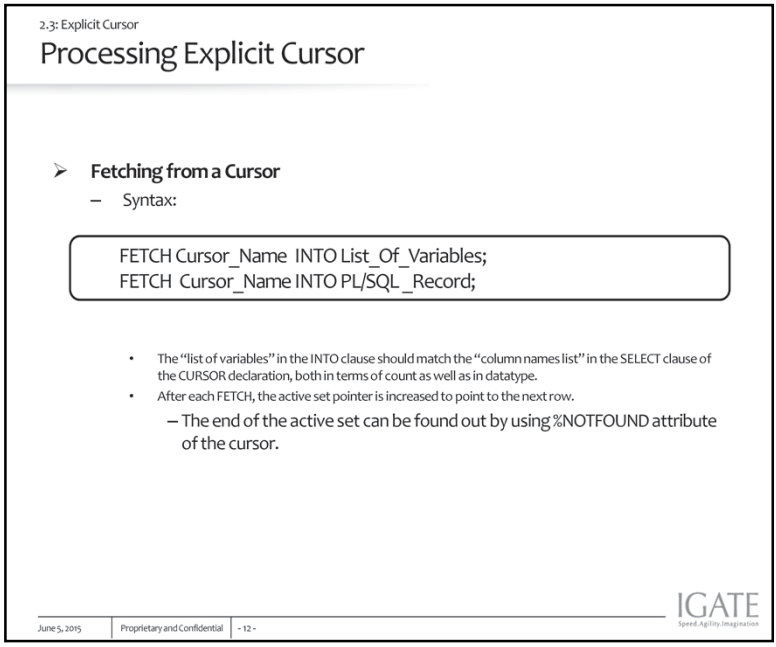
Processing Explicit Cursors: Declaring a Cursor:

- 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.

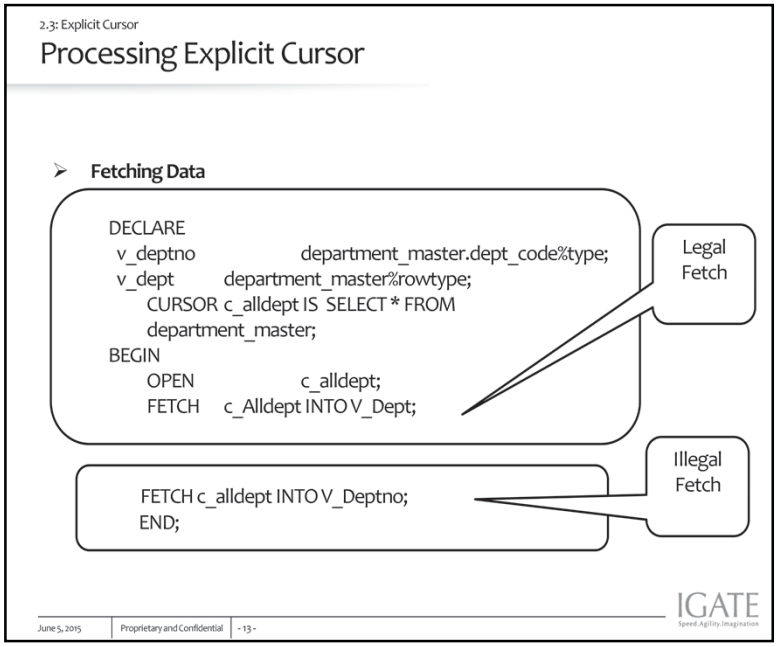


Processing Explicit Cursors: Opening a Cursor:

- 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.



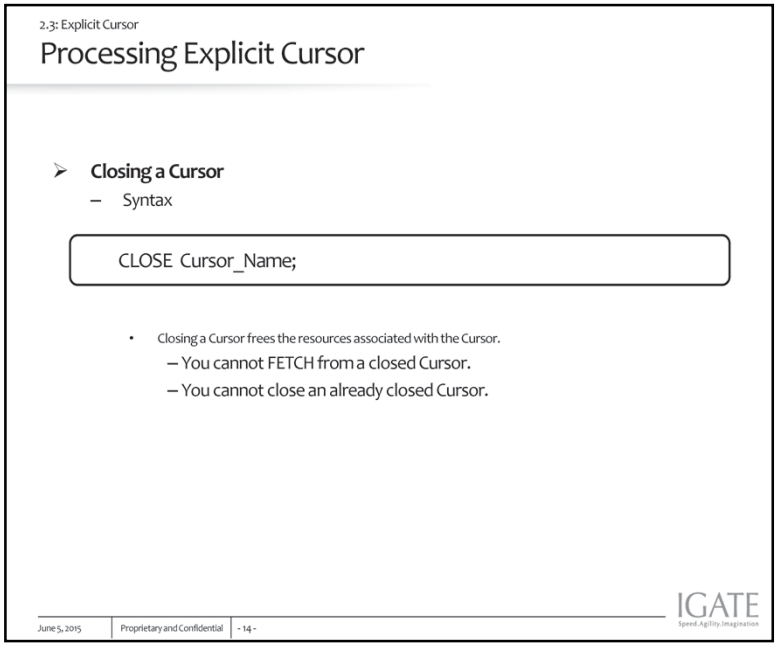
Processing Explicit Cursors: Fetching from Cursor:

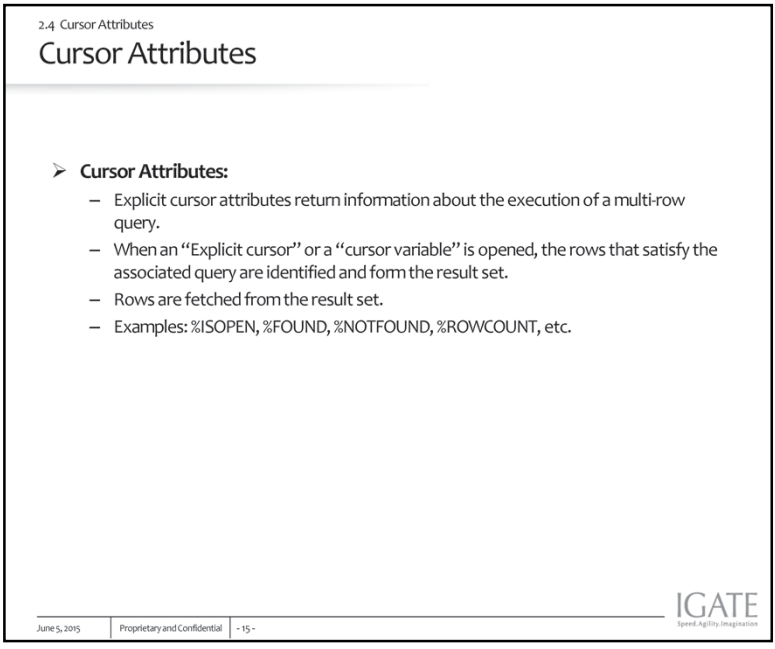


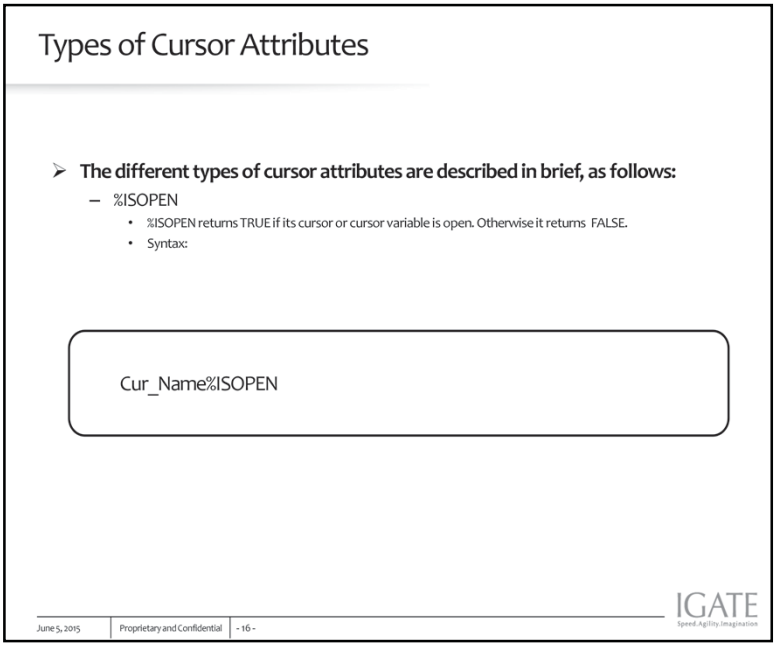
Processing Explicit Cursors: Fetching from Cursor:

- 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.

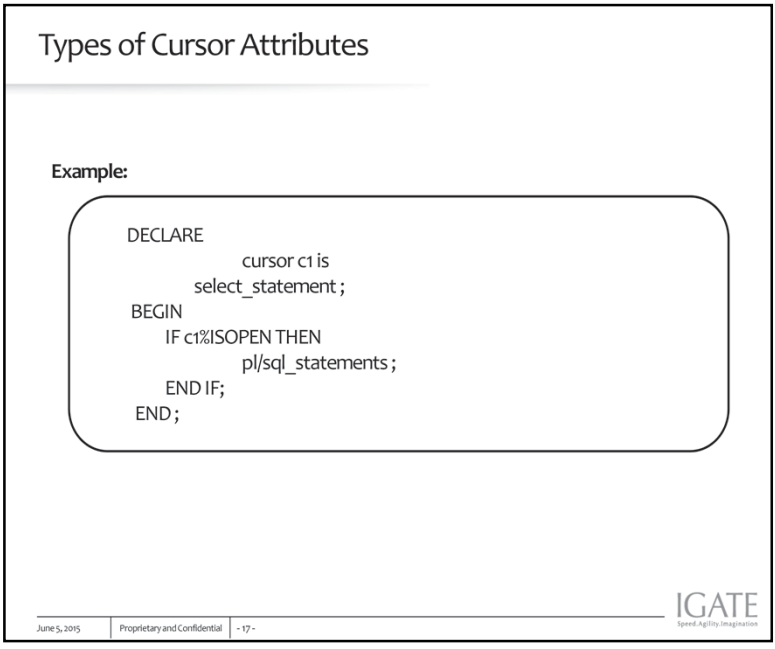

2.3: Explicit Cursor
Processing Explicit Cursor
Closing a Cursor
Syntax
CLOSE Cursor_Name;
Closing a Cursor frees the resources associated with the Cursor.
You cannot FETCH from a closed Cursor.
You cannot close an already closed Cursor.
June 5, 2015
Proprietary and Confidential
- 14 -
IGATE
Speed.Agility.Imagination


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.
June 5, 2015
Proprietary and Confidential
- 15 -
IGATE
Speed.Agility.Imagination



Cursor Attributes: %ISOPEN

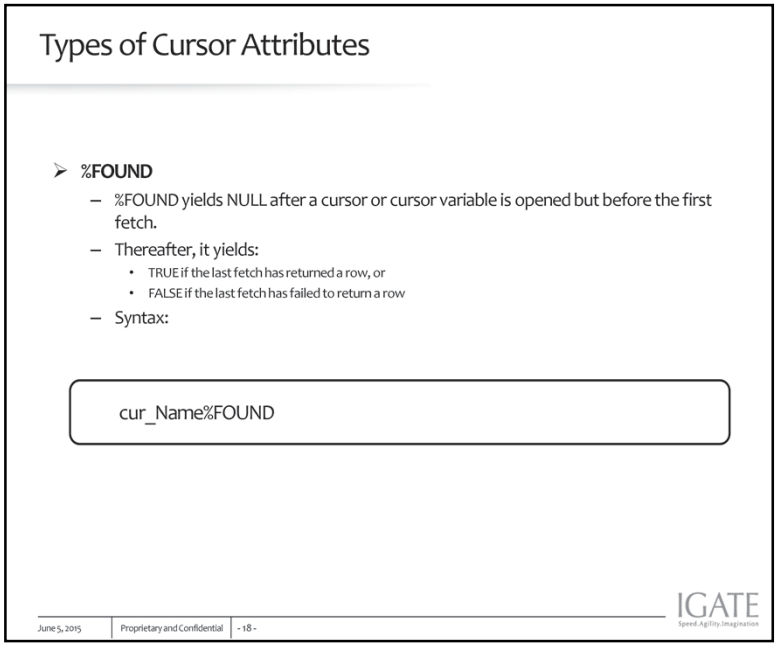
- This attribute is used to check the open/close status of a Cursor.
- If the Cursor is already open, the attribute returns TRUE.
- Oracle closes the SQL cursor automatically after executing its associated SQL statement. As a result, %ISOPEN always yields FALSE for Implicit cursor.



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.



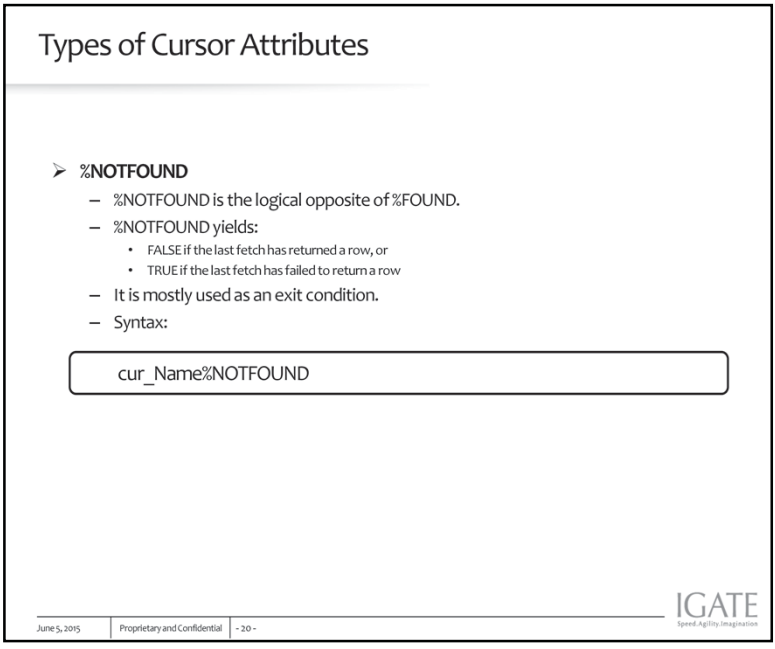
Cursor Attributes: %FOUND:

- 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

Types of Cursor Attributes

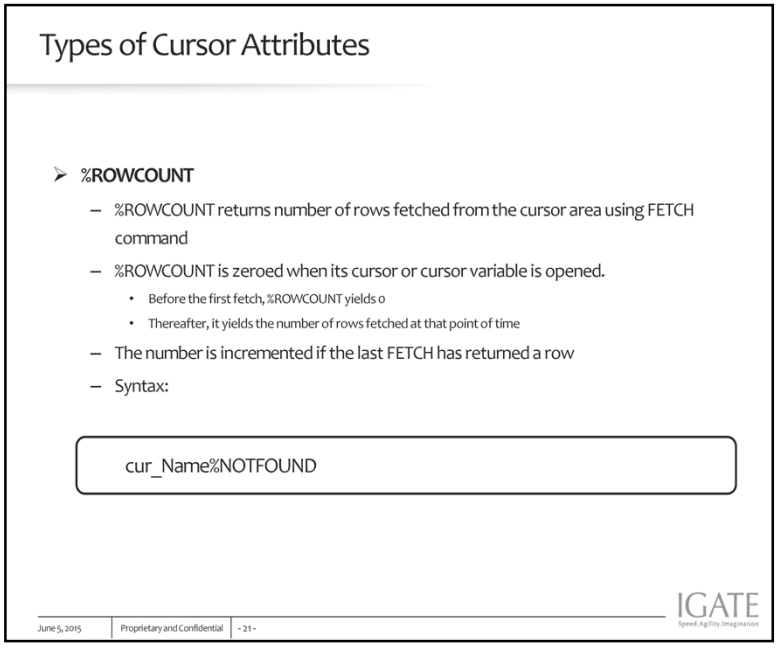
Example:

```
DECLARE section;
   open c1 ;
   fetch c1 into var_list ;
IF c1%FOUND THEN
      pl/sql_statements ;
END IF ;
```



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 ;
```

2.5 Processing cursors

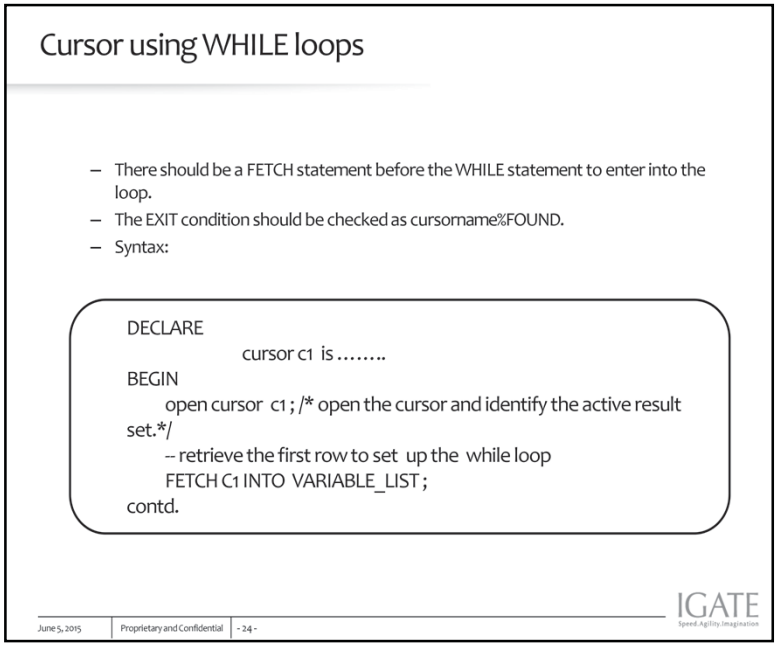
Cursor FETCH loops

- 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

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;
```



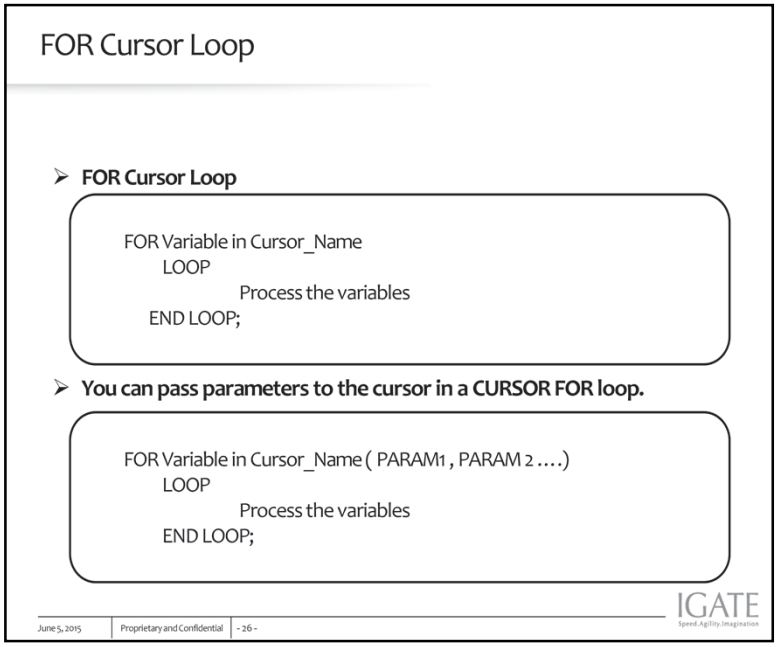
Processing Explicit Cursors: Cursor using WHILE loops:

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.

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;
```



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.

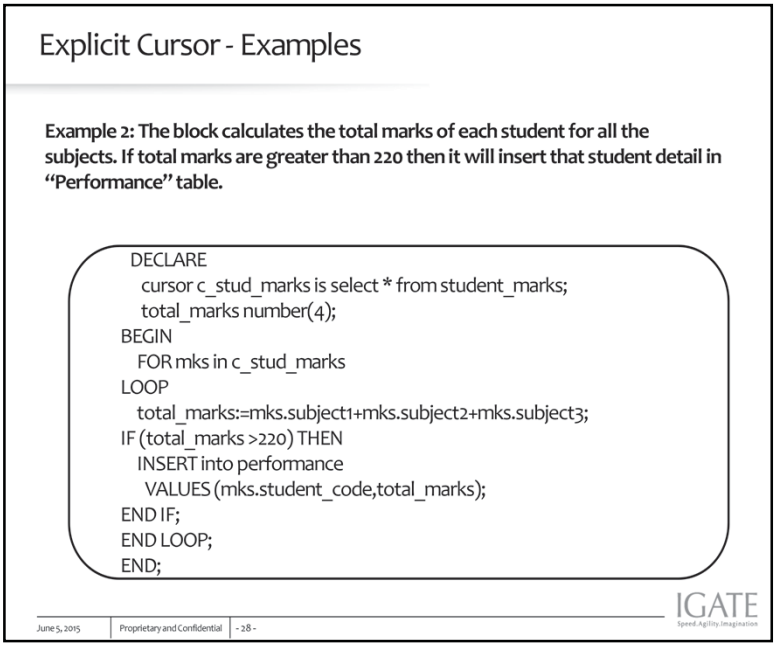
```
DECLARE
CURSOR  C1  IS ……..
BEGIN
-- An implict Open  of the  cursor C1 is done here.
-- Record variable should not be declared in
declaration section
FOR  Record_Variable   IN  C1 LOOP
-- An implicit Fetch is done here
-- Process the fetched rows using variables  and
PL/SQL statements
-- Before 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.

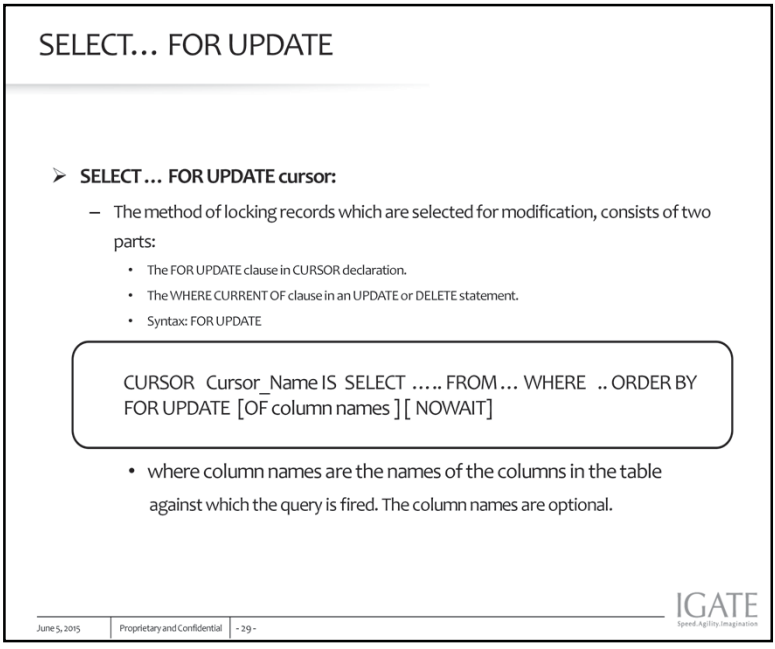
Explicit Cursor - Examples

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;
```

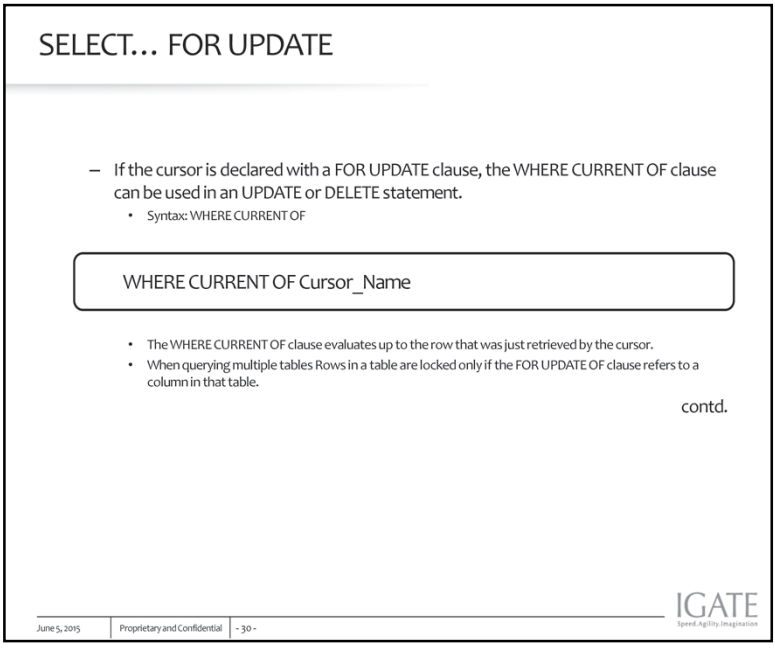


In the above example "Performance" is a user defined table.



Processing Explicit Cursors: Using FOR UPDATE

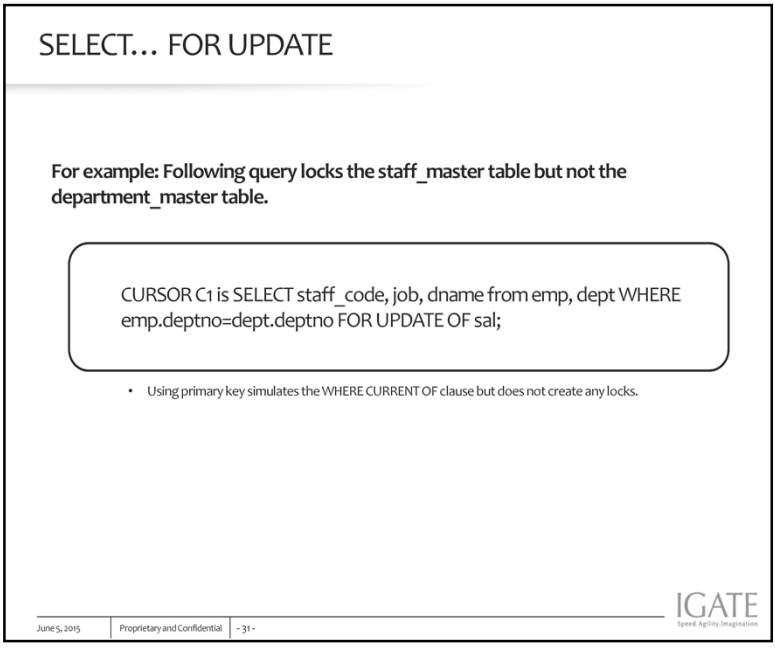
- The FOR UPDATE clause is part of the SELECT statement of the cursor. It is the last clause of the SELECT statement after the ORDER BY clause (if present).
- Normally, a SELECT operation does not lock the rows being accessed. This allows others to change the data selected by the SELECT statement. Besides, at OPEN time the active set consists of changes which were committed. Any changes made after OPEN even if they are committed, are not reflected in the active result set unless the cursor is reopened. This results in Data Inconsistency.
  - ➢If the FOR UPDATE clause is present, exclusive "row locks" are taken on the rows in the active set.
  - ➢These locks prevent other sessions / users from changing these rows unless the changes are committed.
- If another session / user already has locks on the rows in the active set, then the SELECT FOR UPDATE will wait indefinitely for these locks to be released. This statement will hang the system till the locks are released.
  - ➢To avoid this NOWAIT clause is used.
  - ➢With the NOWAIT clause SELECT FOR UPDATE will not wait for the locks acquired by previous sessions to be released and will return immediately with an ORACLE error.



Processing Cursors: Using WHERE CURRENT OF:

Note:

- When querying multiple tables you can use the FOR UPDATE OF column to confine row locking for a particular table.



Processing Cursors: Using WHERE CURRENT OF (contd.):

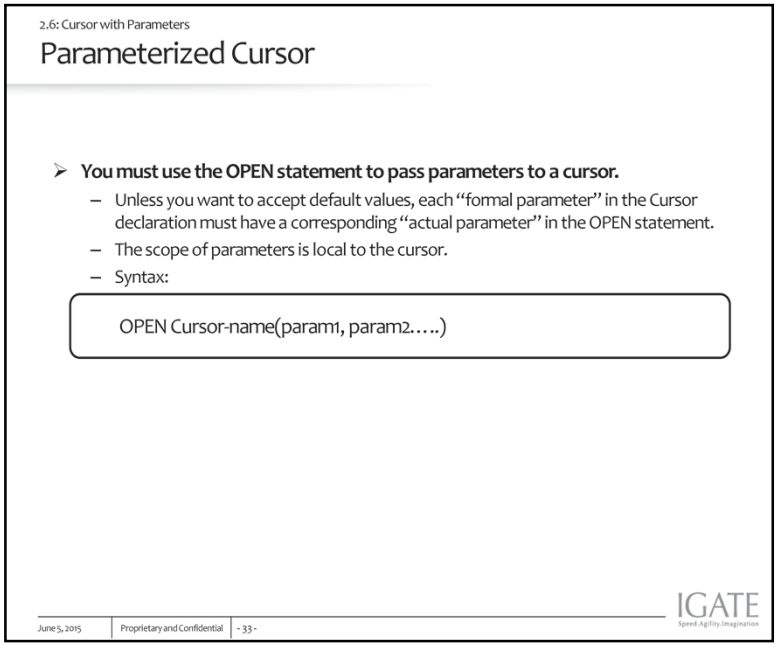
Note:

- If you are using NOWAIT clause, then OF Column_List is essential for syntax purpose.
- Any COMMIT should be done after the processing is over in a CURSOR LOOP which has FOR UPDATE clause. This is because after commit locks will be released, the cursor will become invalid, and further FETCH will result in an error.
- This problem can be solved by using primary key. Using primary key simulates the WHERE CURRENT of clause, however it does not create any locks.

Examples

– To promote professors who earn more than 20000

```
DECLARE
CURSOR c_staff  is SELECT staff_code, staff_master.design_code
 FROM staff_master,designation_master
 WHERE design_name = 'Professor' and staff_sal > 20000
and staff_master.design_code =designation_master.design_code
FOR UPDATE OF design_code NOWAIT;
d_code  designation_master.design_code%type;
BEGIN
    SELECT design_code into d_code FROM designation_master
        WHERE design_name='Director';
      FOR v_rec in c_staff
      LOOP
         UPDATE staff_master SET design_code = d_code
          WHERE current of c_staff;
      END LOOP;
END;
```

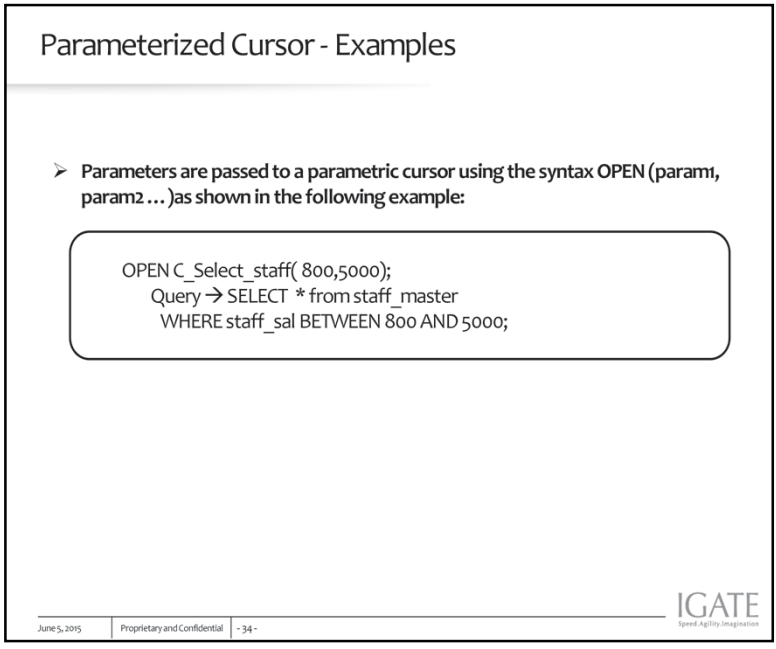


Cursor with Parameters:

- A cursor can take parameters, which can appear in the associated query wherever constants can appear. The formal parameters of a cursor must be IN parameters; they supply values in the query, but do not return any values from the query. You cannot impose the constraint NOT NULL on a cursor parameter.
- Cursor parameters can be referenced only within the query specified in the cursor declaration. The parameter values are used by the associated query when the cursor is opened.

- The scope of parameters is local to the cursor.
- The values of cursor parameters are used by associate queries when the cursor is OPEN.

```
CURSOR C_Select_staff (Low_Sal NUMBER
DEFAULT  0 , High_Sal DEFAULT 10000) IS SELECT  *
from staff_master WHERE  staff_sal  BETWEEN
Low_Sal  AND  High_Sal);
```

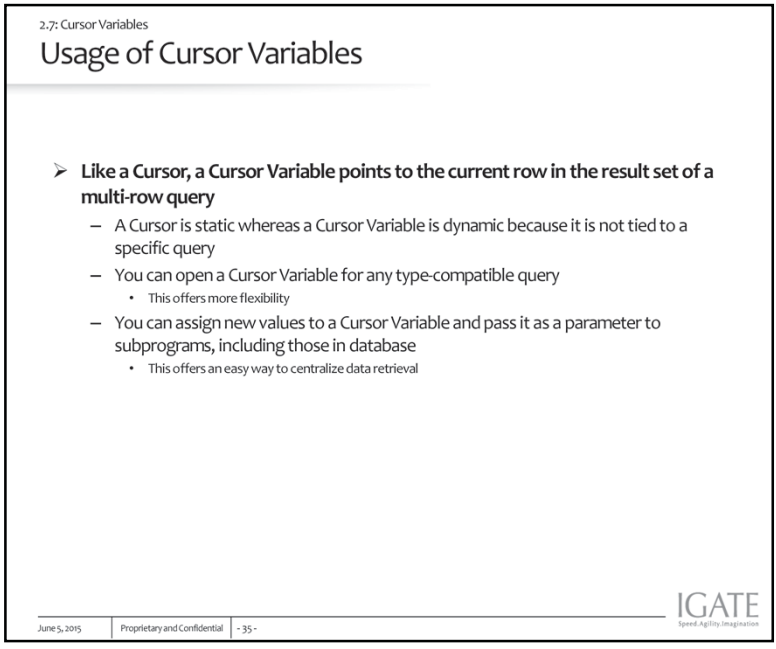


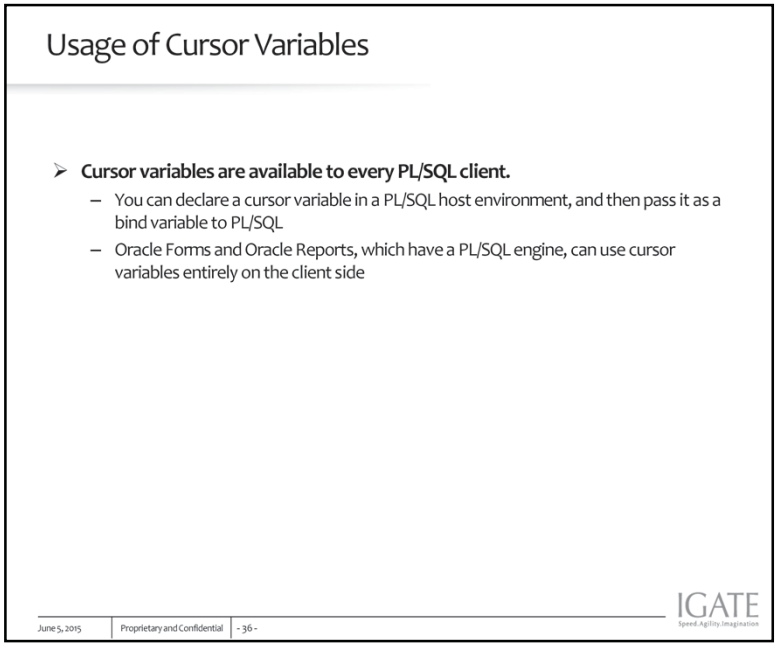
Cursor with Parameters: Passing Parameters to Cursors

- In a FOR CURSOR LOOP parameters are passed using the above syntax:

- Unless you want to accept "default values", each "formal parameter" in the cursor declaration must have a corresponding "actual parameter" in the OPEN statement.
  - Formal parameters having default values need not have corresponding actual values.
  - Each parameter must belong to a datatype which is compatible with the data type of its corresponding formal parameter.

```
FOR  Variable in Cursor_Name (PARAM1 , PARAM 2
….);
FOR V_Get_Det in C_Select_staff( 800,5000)
LOOP
Process the variables
END LOOP;
```

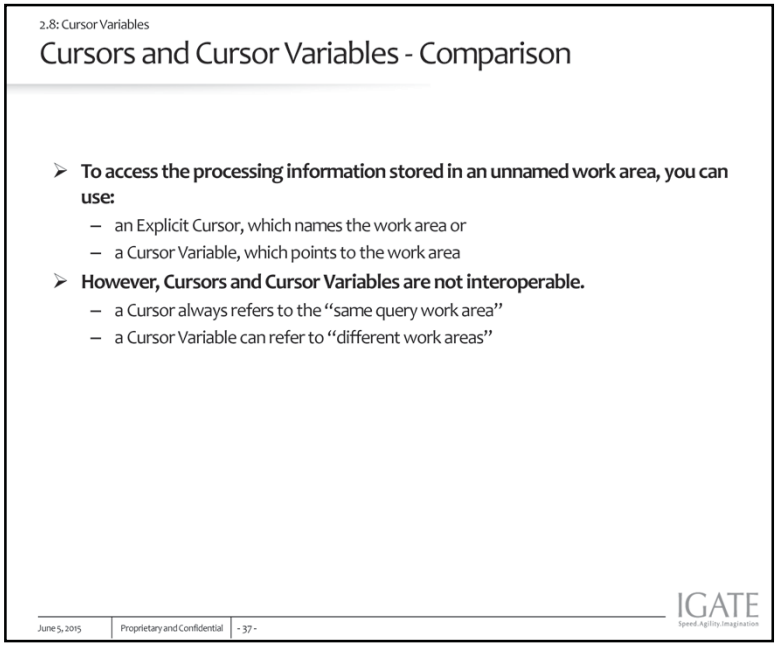

2.7: Cursor Variables
Usage of Cursor Variables
Like a Cursor, a Cursor Variable points to the current row in the result set of a multi-row query
A Cursor is static whereas a Cursor Variable is dynamic because it is not tied to a specific query
You can open a Cursor Variable for any type-compatible query
This offers more flexibility
You can assign new values to a Cursor Variable and pass it as a parameter to subprograms, including those in database
This offers an easy way to centralize data retrieval
June 5, 2015
Proprietary and Confidential
- 35 -
IGATE
Speed.Agility.Imagination



Usage of Cursor Variables:

Note:

- Cursor variables are like 'C' pointers, which hold the memory location (address) of some item instead of the item itself.
- So when you are declaring a Cursor Variable you are creating a "pointer", and not an "item".



Cursors and Cursor Variables:

- In PL/SQL, a pointer has datatype ***REF*** X, where ***REF*** is a short name for *REFERENCE* and X stands for a class of an object. Therefore, a Cursor Variable has datatype REF CURSOR.
- To execute a multi-row query, Oracle opens an unnamed work area that stores processing information.
  - To access the information,
    - you can use an explicit cursor, which names the work area. Or,
    - you can use a cursor variable, which points to the work area.
  - A Cursor always refers to the "same query work area", whereas a Cursor Variable can refer to "different work areas". So, cursors and cursor variables are not interoperable that is, you cannot use one where the other is expected.
- Mainly, Cursor Variables are used "to pass query result sets" between PL/SQL stored subprograms and various clients.
  - Neither PL/SQL nor any of its clients owns a result set.
  - They simply "share a pointer" to the query work area in which the result set is stored.

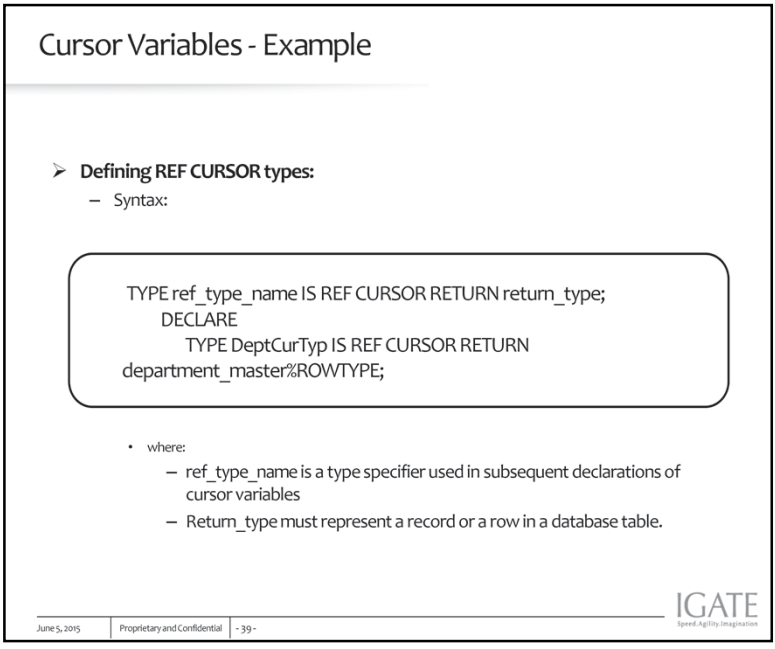
  For example: Oracle Forms application, and Oracle Server can all refer to the same work area.

Cursors and Cursor Variables (contd.):

- A query work area remains accessible as long as any Cursor Variable points to it. Therefore, you can pass the value of a Cursor Variable freely from one scope to another.
  For example: If you pass a "host cursor variable" to a PL/SQL block, the work area to which the Cursor Variable points remains accessible after the block completes.
- If you have a PL/SQL engine on the client side, then calls from client to server impose no restrictions.
  For example: You can declare a Cursor Variable on the client side, open and fetch from it on the server side, and then continue to fetch from it back on the client side. Also, you can reduce network traffic by having a *PL/SQL* block open (or close) several "host cursor variables" in a single round trip.

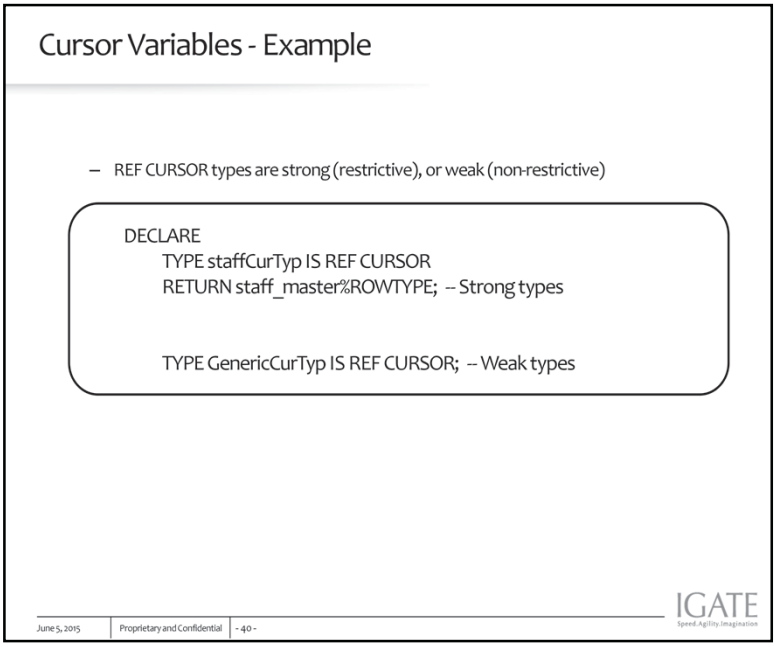
Creating Cursor Variables:

- To create cursor variables, two steps are required in the same sequence:
  1. Define a *REF CURSOR* type
  2. Declare cursor variables of that type
- You can define REF CURSOR types in any PL/SQL block, subprogram, or package.



Cursors and Cursor Variables: Defining REF CURSOR types:

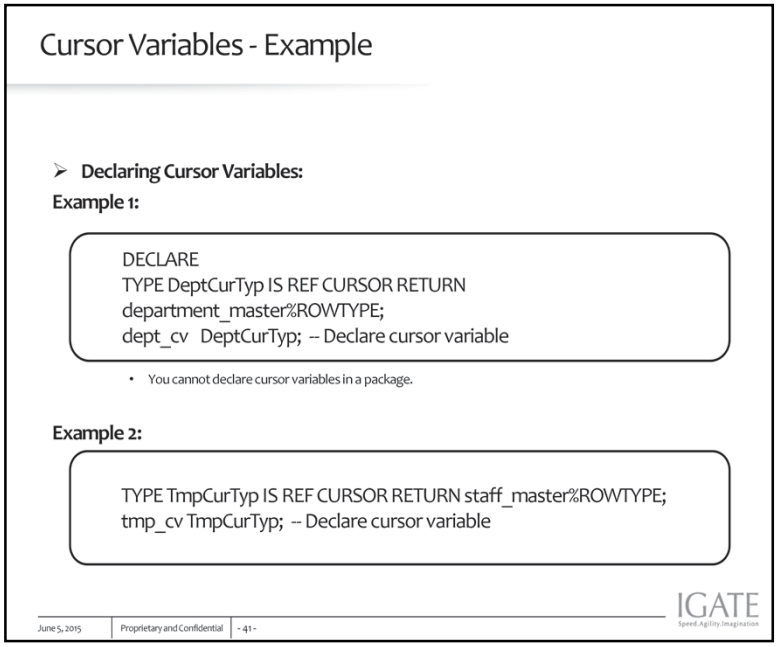
- **Example:** In the example shown on the slide , you specify a Return Type that represents a row in the database table department_master:



Cursors and Cursor Variables: Defining REF CURSOR types (contd.):

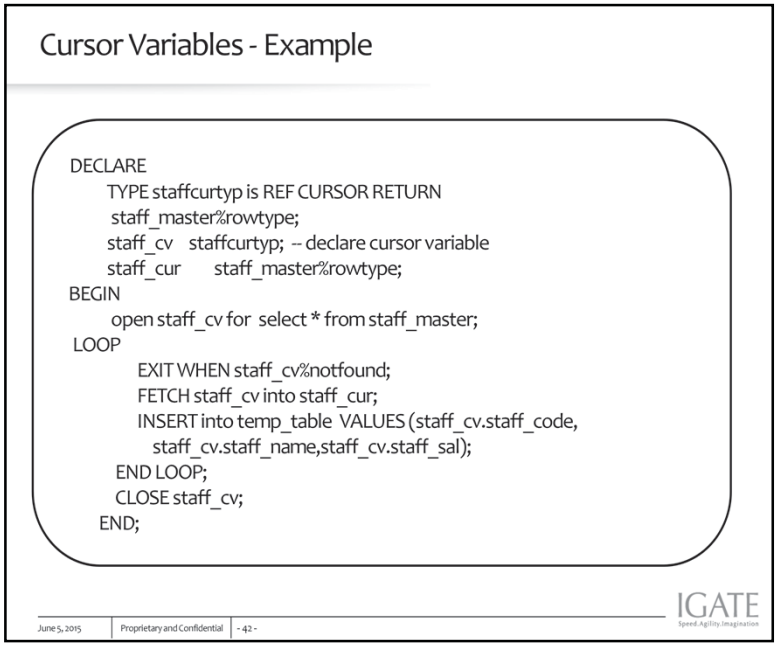
Note:

- A strong REF CURSOR type definition specifies a Return Type.
- However, a weak definition does not specify a Return Type.
- Strong REF CURSOR types are less error prone because the PL/SQL compiler lets you associate a strongly typed Cursor Variable only with type-compatible queries.
- However, weak REF CURSOR types are more flexible because the compiler lets you associate a weakly typed cursor variable with any query.



Cursors and Cursor Variables: Declaring Cursor Variable:

- As shown in examples shown on the slide, in the RETURN clause of a REF CURSOR type definition, you can use %ROWTYPE to specify a record type that represents a row returned by a strongly (not weakly) typed cursor variable.



- In the example shown on the slide, using a cursor variable we are fetching data from a table and inserting it in a temp_table.

Summary

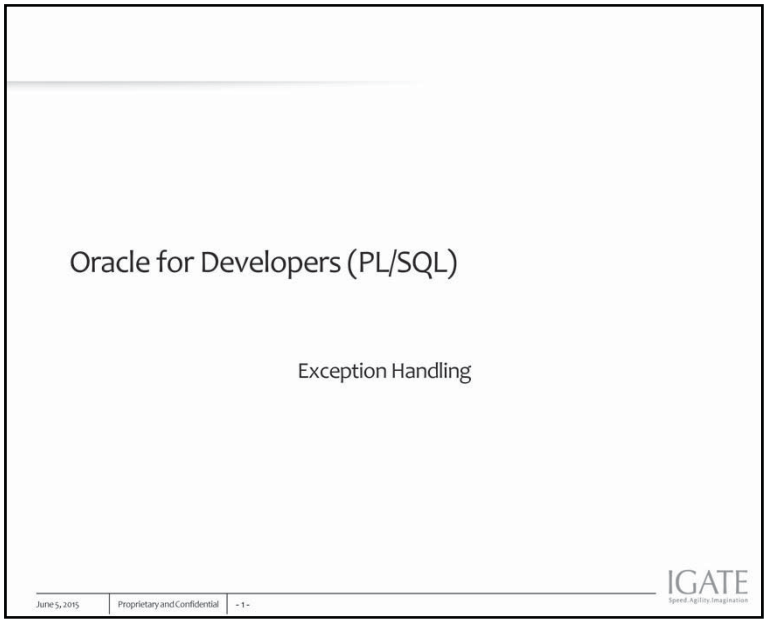
- **In this lesson, you have learnt:**
  - Cursor is a "handle" or "name" for a private SQL area
  - Implicit cursors are declared for queries that return only one row
  - Explicit cursors are declared for queries that return more than one row
  - Like a Cursor, a Cursor Variable points to the current row in the result set of a multi-row query
  - However, Cursors and Cursor Variables are not interoperable

Review – Questions

- **Question 1: A "Cursor" is static whereas a "Cursor Variable" is dynamic because it is not tied to a specific query.**
  - True / False
- **Question 2: %COUNT returns number of rows fetched from the cursor area by using FETCH command.**
  - True / False

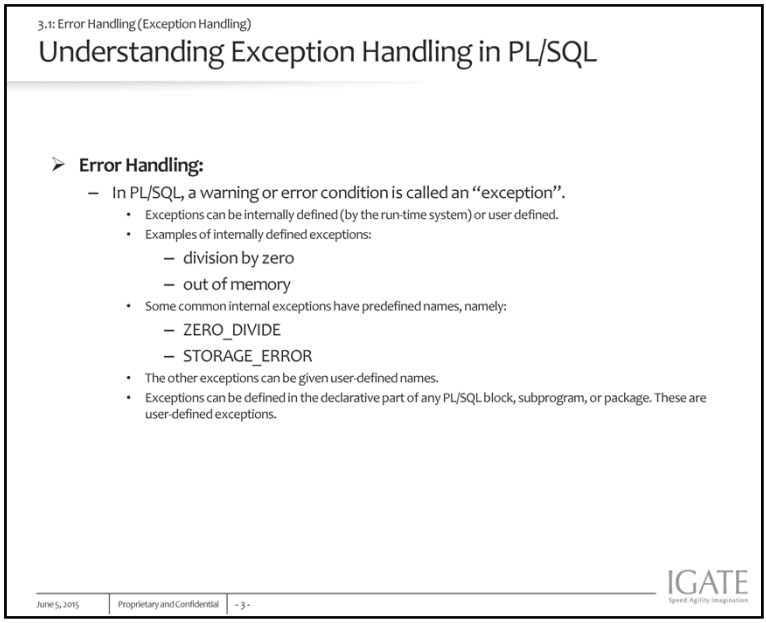
Review – Questions

- **Question 3: Implicit SQL cursor is opened or closed by the program.**
  - True / False
- **Question 4: A ___ specifies a Return Type.**
- **Question 5: PL/SQL provides a shortcut via a ___ Loop, which implicitly handles the cursor processing.**


Oracle for Developers (PL/SQL)
Exception Handling
June 5, 2015
Proprietary and Confidential
- 1 -
IGATE
Speed. Agility. Imagination

Lesson Objectives

- **To understand the following topics:**
  - Error Handling
  - Declaring Exceptions
    - Predefined Exceptions
    - Numbered Exceptions
    - User Defined Exceptions
  - Raising Exceptions
  - Control passing to Exception Handlers
  - RAISE_APPLICATION_ERROR

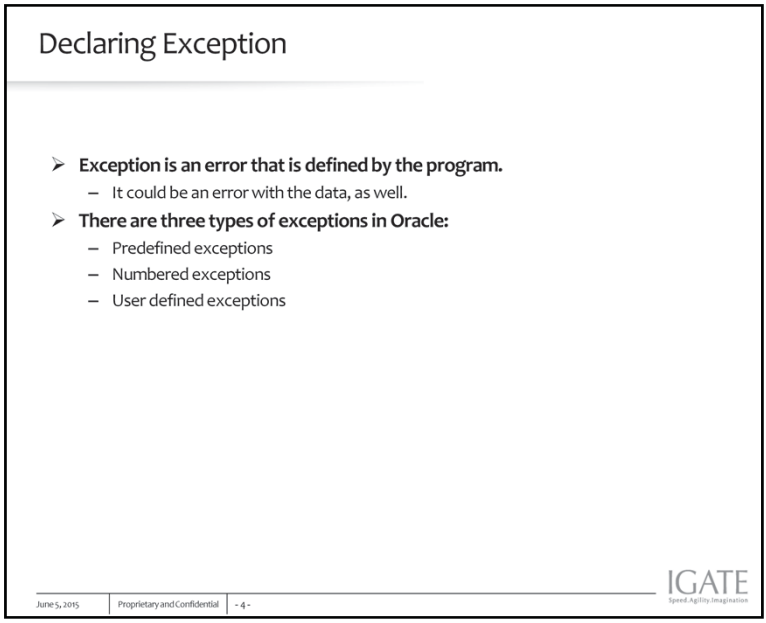


Error Handling:

- 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".

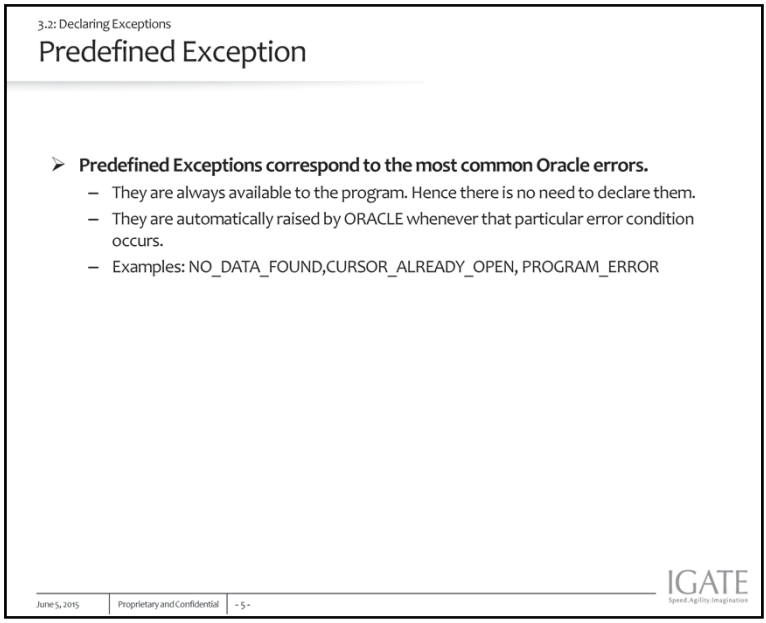
Types of Errors in PL/SQL

- **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.



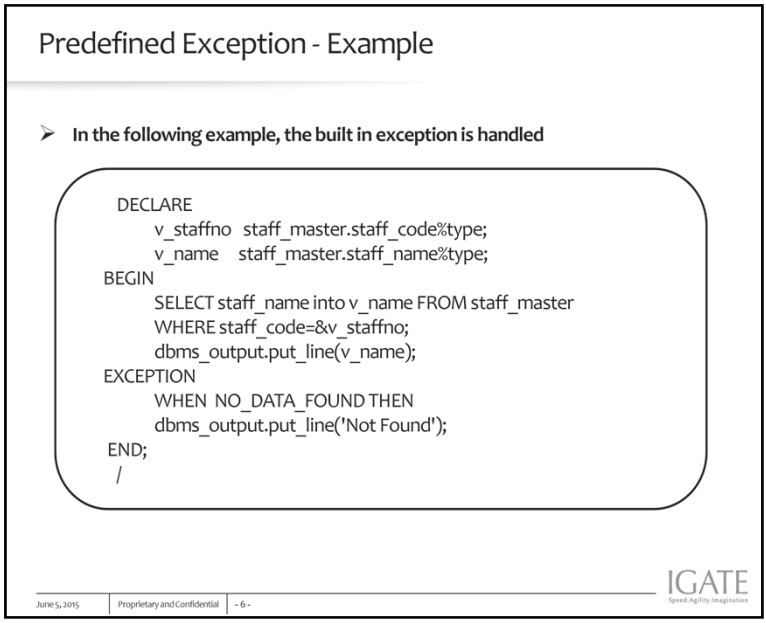
Declaring Exceptions:

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



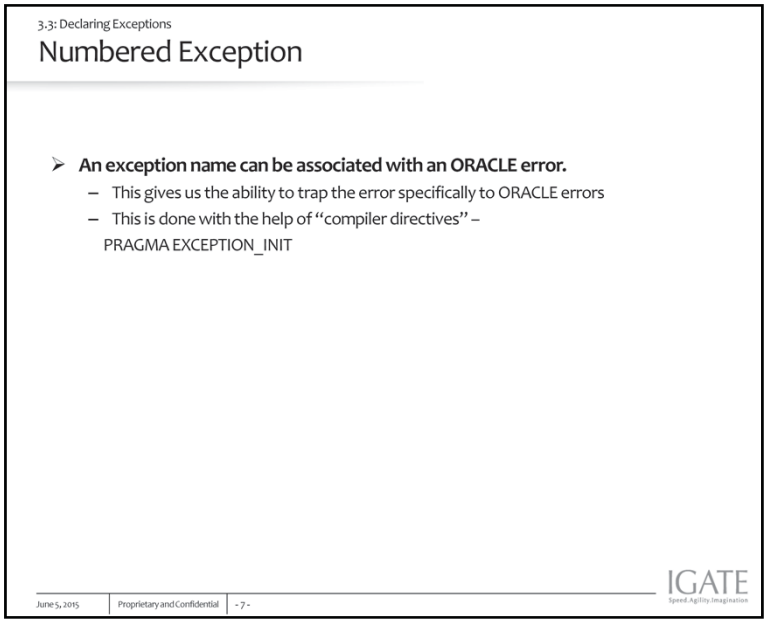
Predefined Exceptions:

- 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.



Numbered Exception:
The Numbered Exceptions are Oracle errors bound to a user defined exception name.

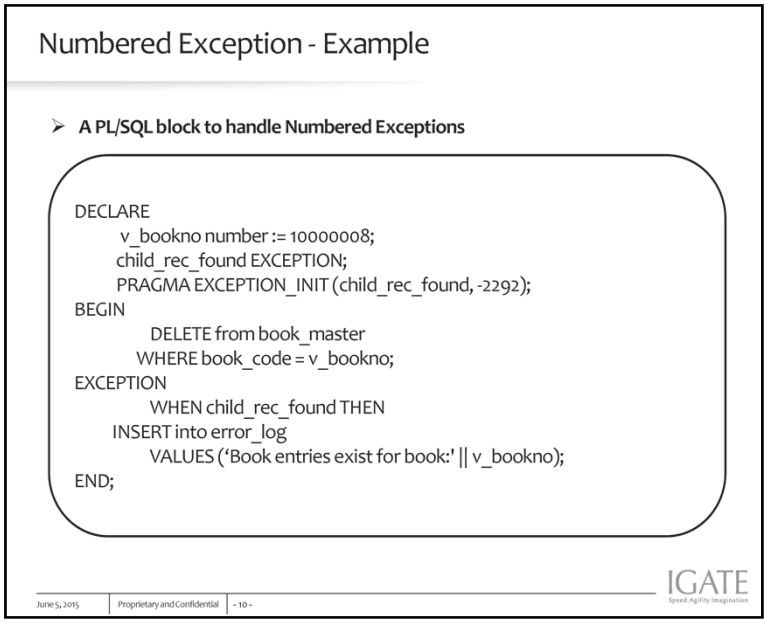
Numbered Exception

- **PRAGMA EXCEPTION_INIT:**
  - A PRAGMA is a compiler directive that is processed at compile time, not at run time. It is used to name an exception.
  - In PL/SQL, the PRAGMA EXCEPTION_INIT tells the compiler to associate an exception name with an Oracle error number.
    - This arrangement lets you refer to any internal exception(error) by name, and to write a specific handler for it.
  - When you see an error stack, or sequence of error messages, the one on top is the one that you can trap and handle.

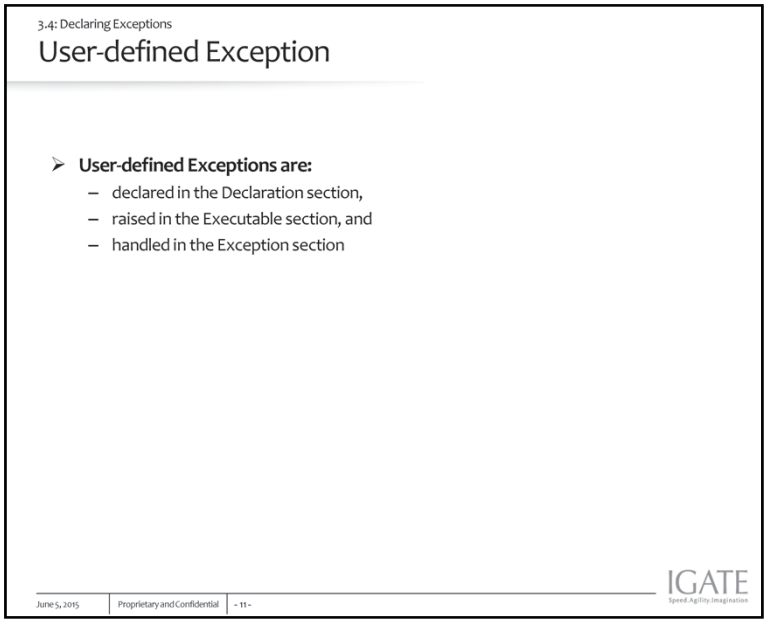
Numbered Exception

- User defined exceptions can be named with error number between -20000 and -20999.
- The naming is declared in Declaration section.
- It is valid within the PL/SQL blocks only.
- Syntax is:

```
PRAGMA EXCEPTION_INIT(Exception Name,Error_Number);
```



If a user tries to delete record from the parent table wherein child records exist an error is raised by Oracle. We would want to handle this error through the PL/SQL block which is deleting records from a parent table. The example on the slide demonstrates this. In the PL/SQL block we are binding the constraint exception raised by Oracle to user defined exception name.
All oracle errors are negative i.e prefixed with a minus symbol. In the example we are mapping error-2292 which occurs when referential integrity rule is violated.



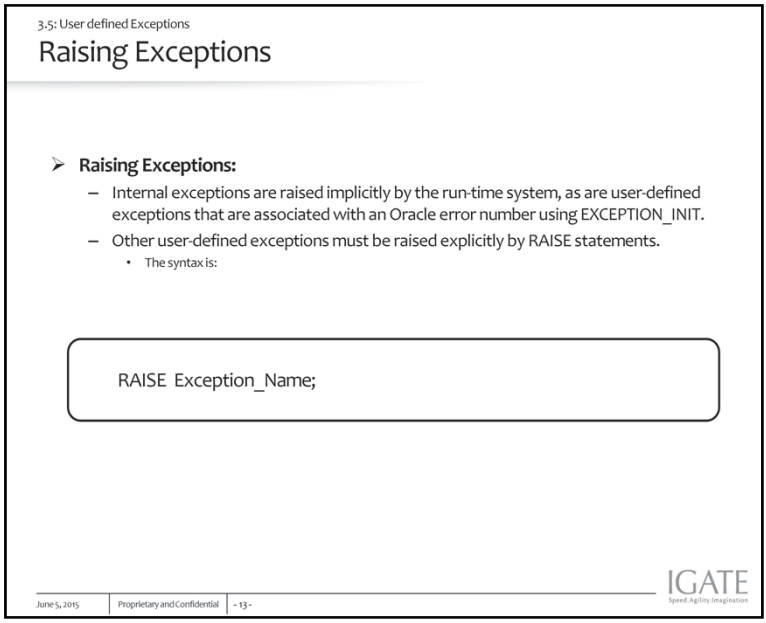
User-Defined Exceptions:

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

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;
```



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

Raising Exceptions - Example

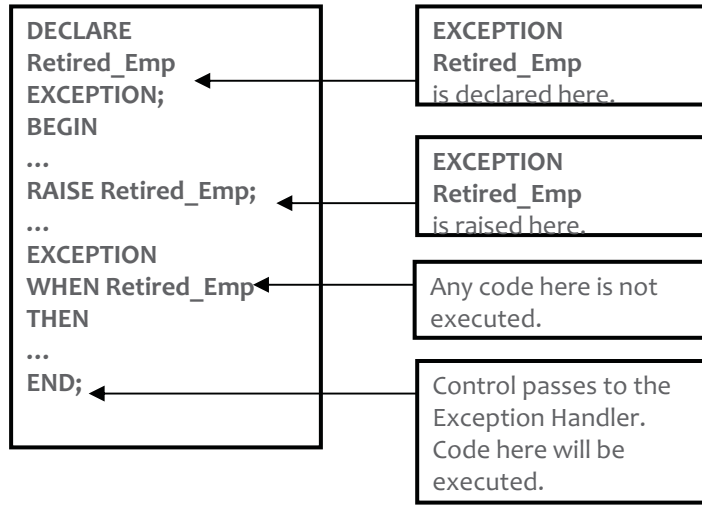
- **An exception is defined and raised as shown below:**

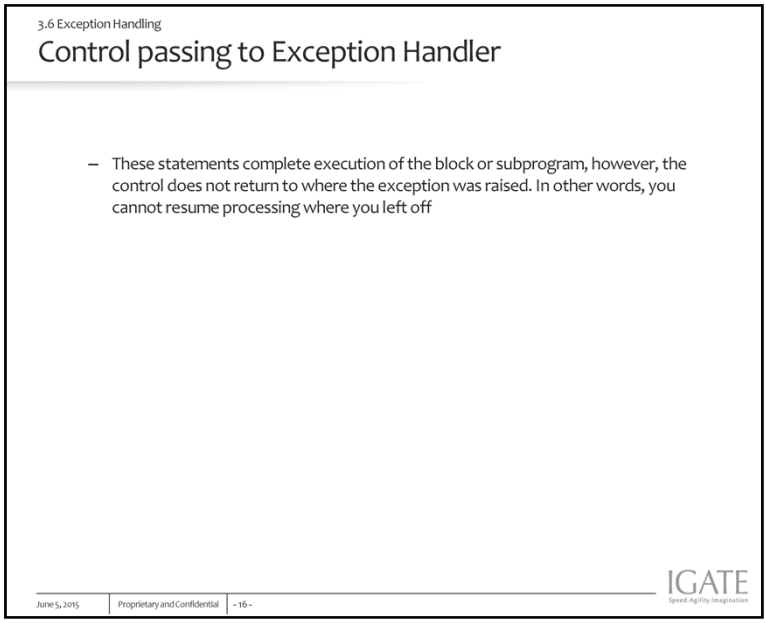
```
DECLARE
      ...
      retired_emp EXCEPTION;
BEGIN
      pl/sql_statements;
      if error_condition then
      RAISE retired_emp;
      pl/sql_statements;
EXCEPTION
      WHEN retired_emp THEN
      pl/sql_statements;
END;
```

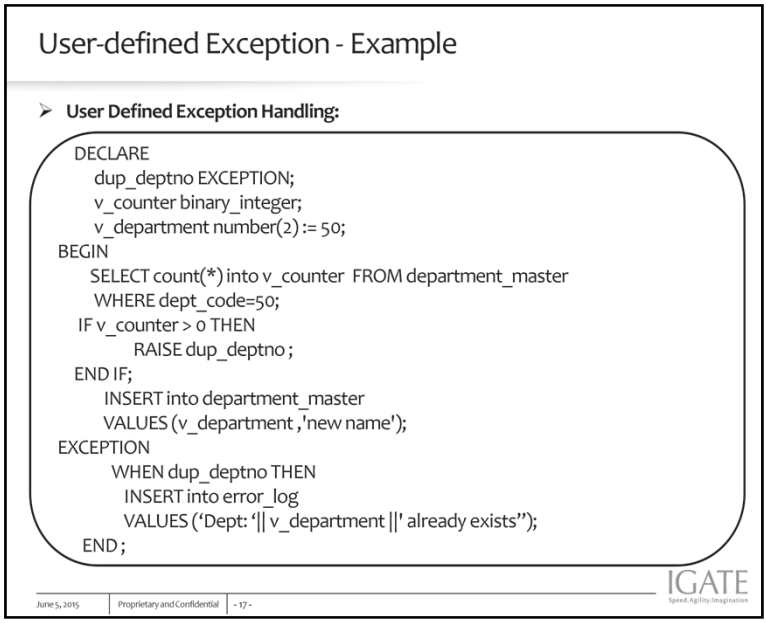
Control passing to Exception Handler

- **Control passing to Exception Handler :**
  - When an exception is raised, normal execution of your PL/SQL block or subprogram stops, and control passes to its exception-handling part.
  - To catch the raised exceptions, you write "exception handlers".
    - Each exception handler consists of a WHEN clause, which specifies an exception, followed by a sequence of statements to be executed when that exception is raised.

Control passing to Exception Handler:




3.6 Exception Handling
Control passing to Exception Handler
– These statements complete execution of the block or subprogram, however, the control does not return to where the exception was raised. In other words, you cannot resume processing where you left off
June 5, 2015
Proprietary and Confidential
- 16 -
IGATE
Speed.Agility.Imagination



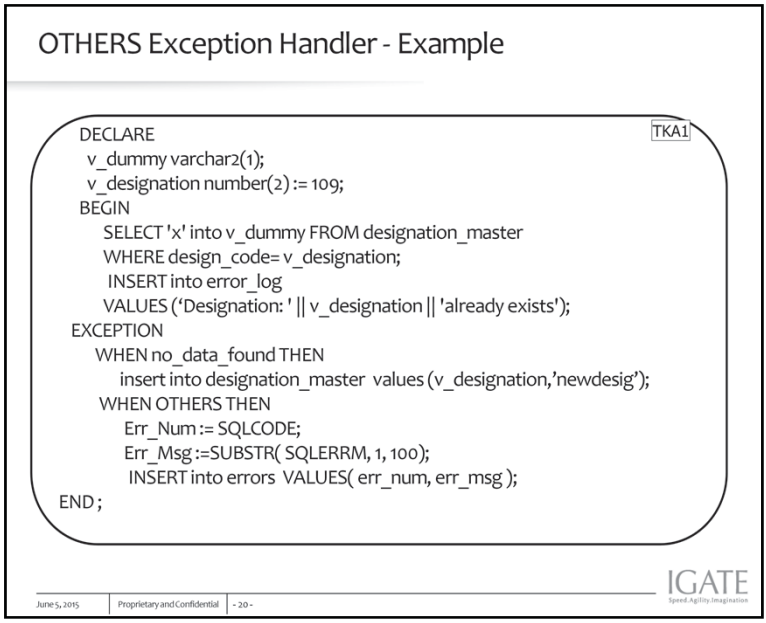
The example on the slide demonstrates user-defined exceptions. It checks for department no value to be inserted in the table. If the value is duplicated it will raise an exception.

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.

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.



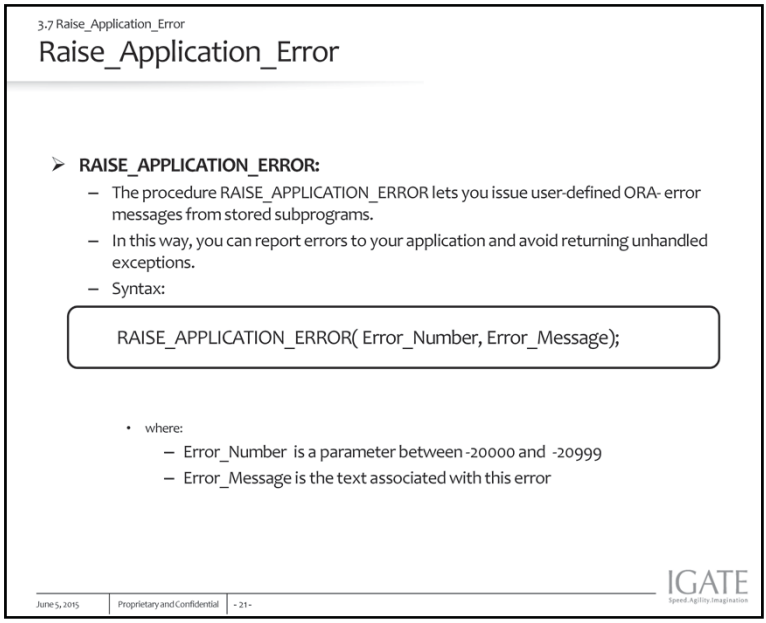
The example on the slide uses OTHERS Exception handler. If the exception that is raised by the code is not NO_DATA_FOUND, then it will go to the OTHERS exception handler since it will notice that there is no appropriate exception handler defined. Also observe that the values of SQLCODE and SQLERRM are assigned to variables defined in the block.

Slide 20

TKA1 As this code is having comilation problem . Pls find the correct code .

```
DECLARE
 v_dummy varchar2(1);
 v_designation NUMBER(3):=356;
 Err_Num NUMBER(8);
 Err_Msg VARCHAR2(100);
 BEGIN
 SELECT 'x' into v_dummy FROM designation_master
 WHERE design_code=v_designation;
 INSERT into error_log
 VALUES('Designation:'||v_designation||'already exists');
 EXCEPTION
 WHEN no_data_found THEN
    insert into designation_master VALUES(v_designation,'newdesig');
 WHEN OTHERS THEN
   Err_Num:=SQLCODE;
   Err_Msg := SUBSTR(SQLERRM,1,100);
 INSERT into errors VALUES(err_num,err_msg);
 END;
```

Tanmaya K Acharya, 1/29/2015



Raise Application Error:
The built-in function RAISE_APPLICATION_ERROR is used to create our own error messages, which can be more descriptive and user friendly than Exception Names.

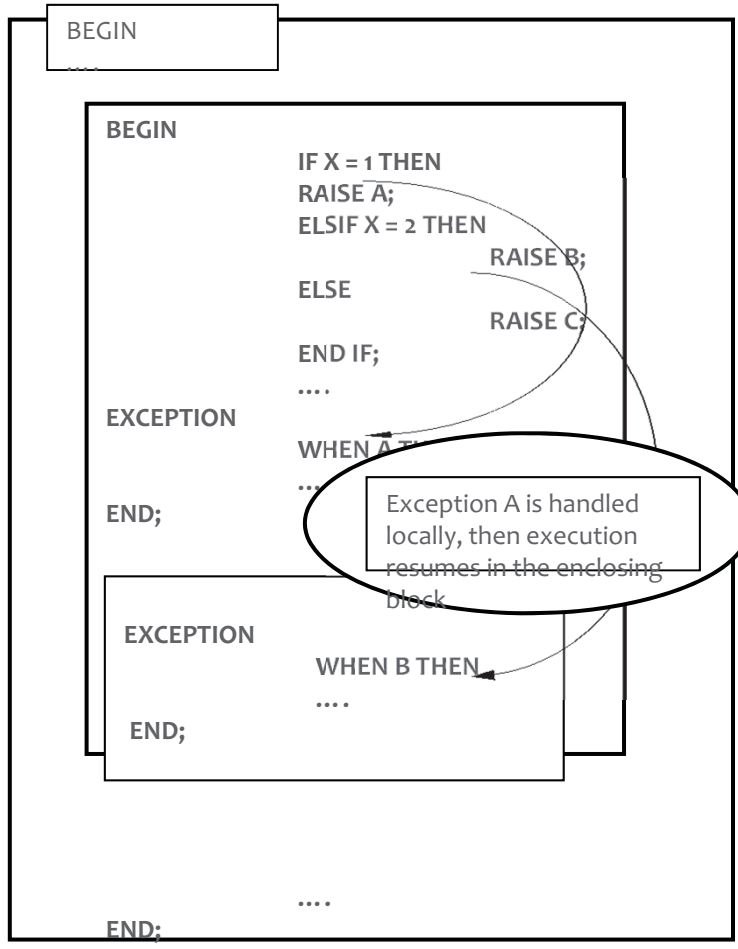
Raise_Application_Error - Example

- **Here is an example of Raise Application Error:**

```
DECLARE
      /* VARIABLES */
BEGIN
      ……..
      ……..
EXCEPTION
      WHEN OTHERS THEN
      -- Will transfer the error to the calling environment
      RAISE_APPLICATION_ERROR( -20999 ,'Contact              DBA');
END ;
```

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.):

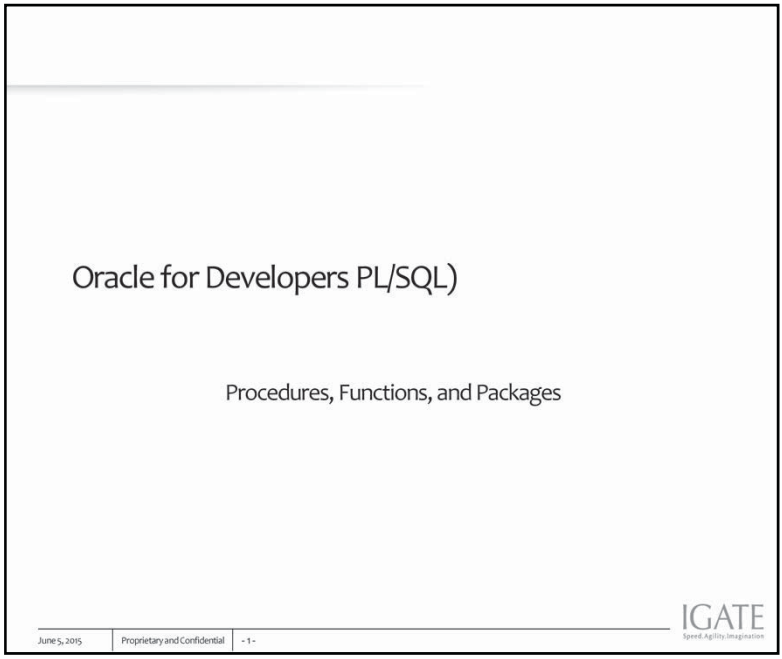
```
BEGIN
--------
./* PL/SQL statements */
BEGIN
SELECT ……..
WHEN  NO_DATA_FOUND  THEN
-- Process  the error for
NO_DATA_FOUND
END;

/* 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
    - User-defined Exceptions
    - Predefined Exceptions
  - Control passing to Exception Handler
  - OTHERS exception handler
  - Association of Exception name to Oracle errors
  - RAISE_APPLICATION_ERROR procedure

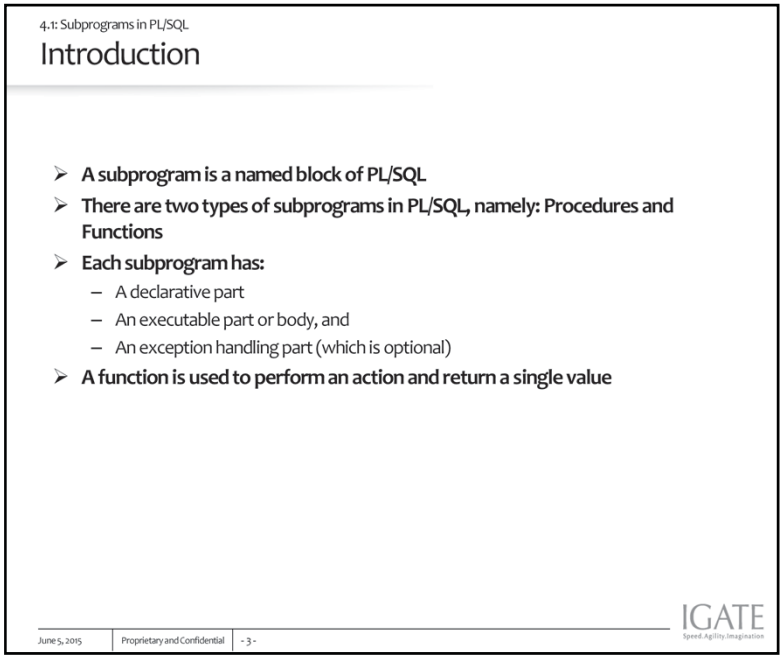
Review – Questions

- Question 1: The procedure ___ lets you issue user-defined ORA-error messages from stored subprograms.
- Question 2: The ___ tells the compiler to associate an exception name with an Oracle error number.
- Question 3: ___ returns the current error code. And ___ returns the current error message text.


Oracle for Developers PL/SQL)
Procedures, Functions, and Packages
June 5, 2015
Proprietary and Confidential
- 1 -
IGATE
Speed. Agility. Imagination

Lesson Objectives

- **To understand the following topics:**
  - Subprograms in PL/SQL
  - Anonymous blocks versus Stored Subprograms
  - Procedure
    - Subprogram Parameter modes
  - Functions
  - Packages
    - Package Specification and Package Body
  - Autonomous Transactions



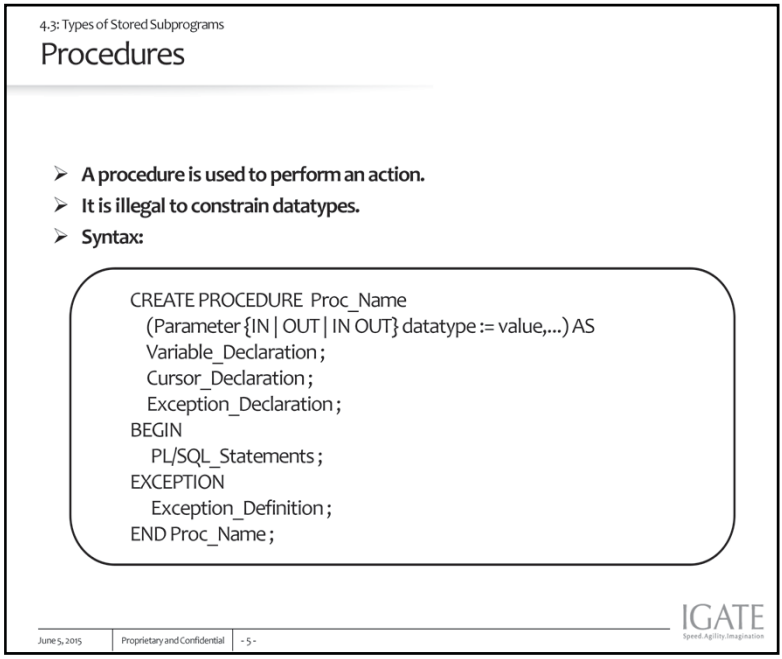
Subprograms in PL/SQL:

- 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: Subprograms in PL/SQL

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.



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.

4.3: Procedures

Subprogram Parameter Modes

IN	OUT	IN OUT
The default	Must be specified	Must be specified
Used to pass values to the procedure.	Used to return values to the caller.	Used to pass initial values to the procedure and return updated values to the caller.
Formal parameter acts like a constant.	Formal parameter acts like an uninitialized variable.	Formal parameter acts like an uninitialized variable.
Formal parameter cannot be assigned a value.	Formal parameter cannot be used in an expression, but should be assigned a value.	Formal parameter should be assigned a value.
Actual parameter can be a constant, literal, initialized variable, or expression.	Actual parameter must be a variable.	Actual parameter must be a variable.
Actual parameter is passed by reference (a pointer to the value is passed in).	Actual parameter is passed by value (a copy of the value is passed out) unless NOCOPY is specified.	Actual parameter is passed by value (a copy of the value is passed in and out) unless NOCOPY is specified.

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;
```

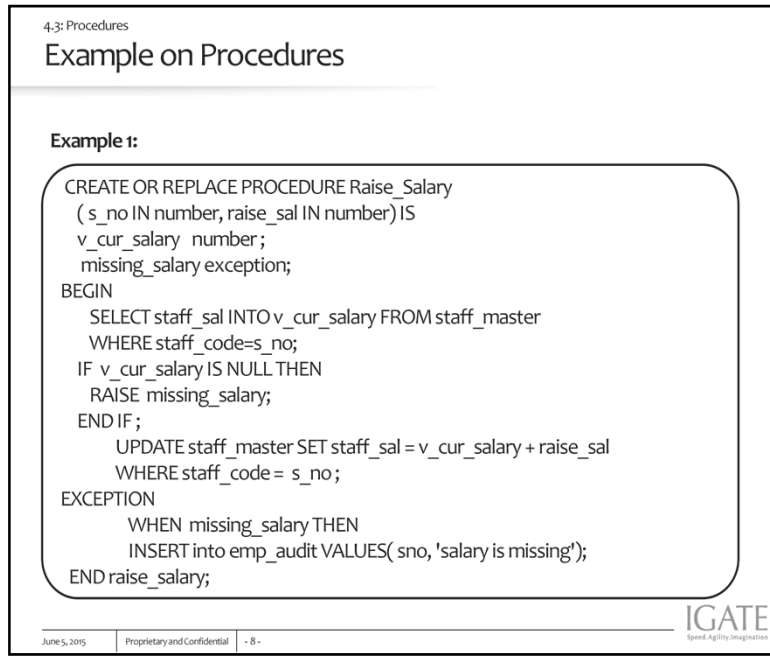
Subprogram Parameter Modes (contd.):
Examples:

Example 2:

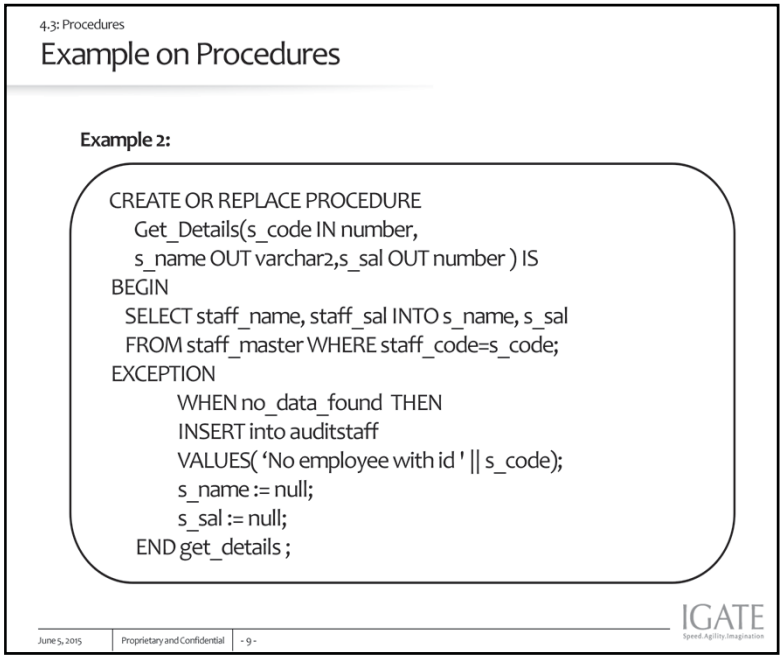
```
SQL > SET SERVEROUTPUT ON
SQL > CREATE OR REPLACE PROCEDURE PROC1 AS
      2    BEGIN
      3          DBMS_OUTPUT.PUT_LINE('Hello from procedure … ');
      4    END;
      5    /
Procedure created.
SQL > EXECUTE PROC1
Hello from procedure …
PL/SQL procedure successfully created.
```

Example 3:

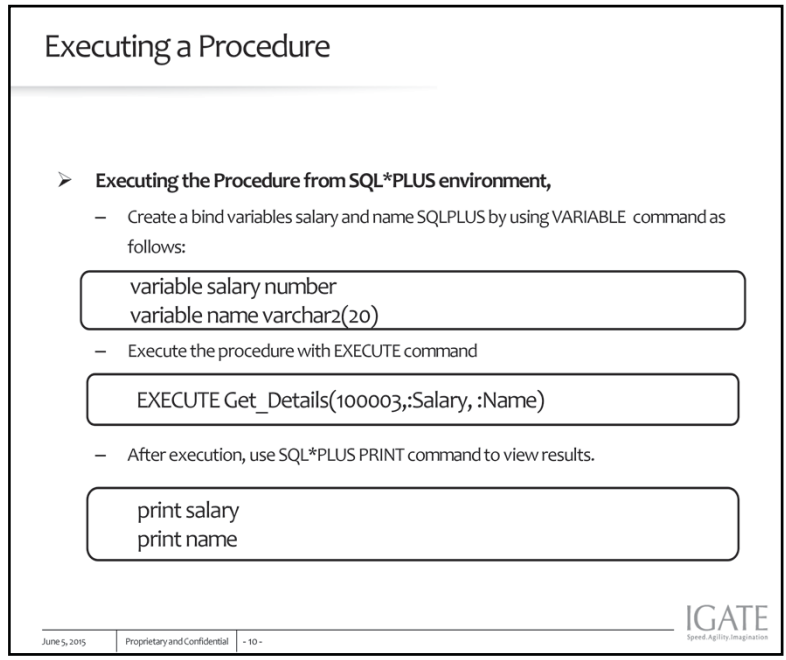
```
SQL > CREATE OR REPLACE          PROCEDURE PROC2
      2       (N1 IN NUMBER, N2 IN NUMBER, TOT OUT NUMBER) IS
      3    BEGIN
      4       TOT := N1 + N2;
      5    END;
      6    /

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 ;
```

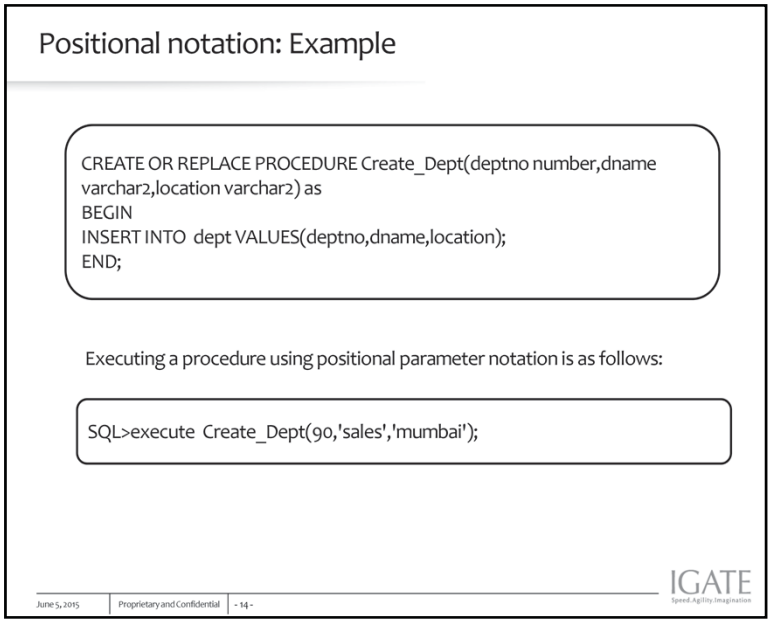
```
BEGIN
Create_Dept( 50);
-- Actual  call will be  Create_Dept ( 50, 'TEMP',
'TEMP')

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');
```

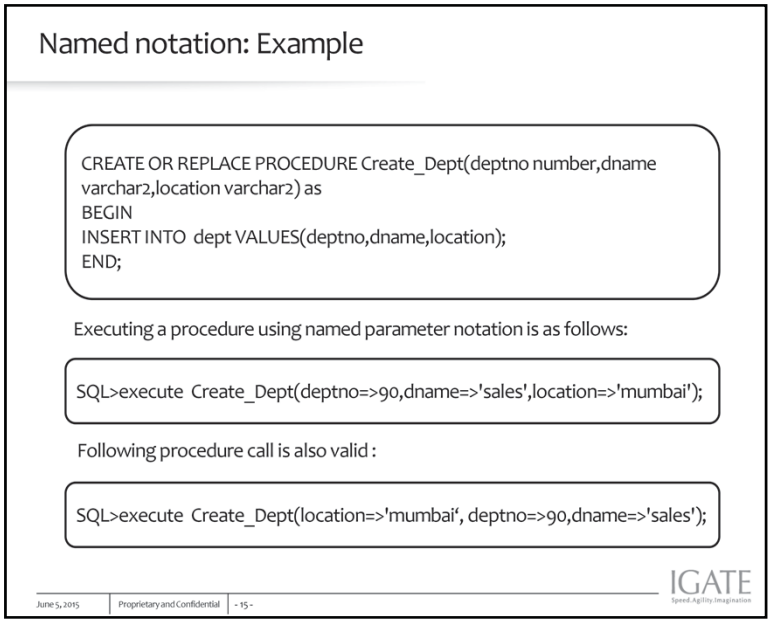


Positional Notation :

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.

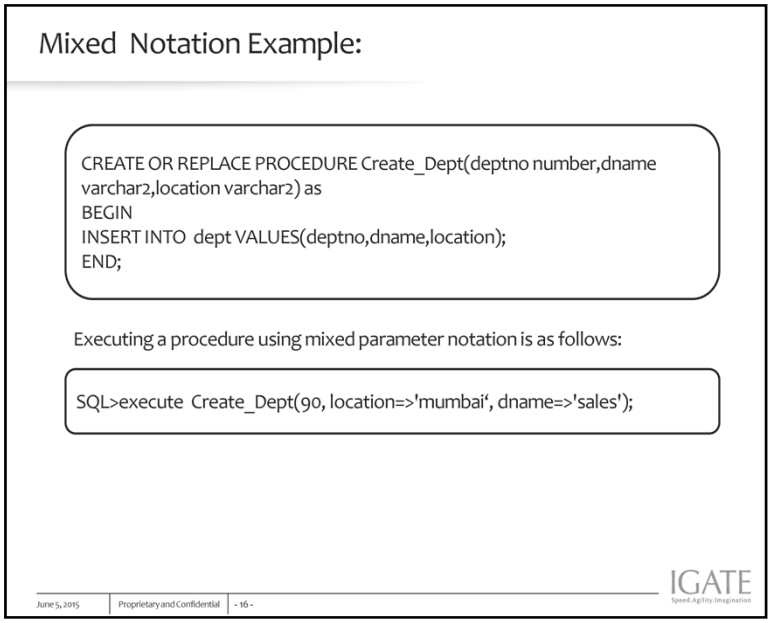
You must change your code if the procedure's parameter list changes



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: Types of Stored Subprograms

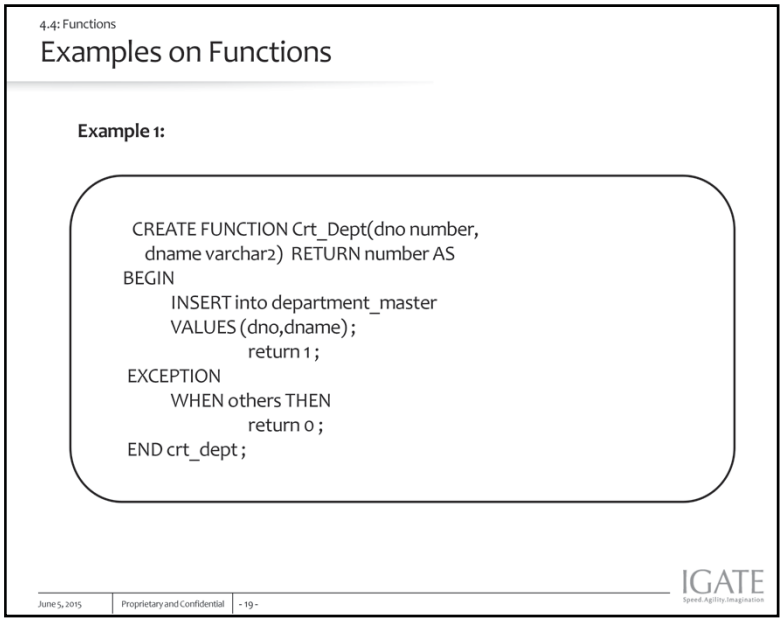
Functions

- **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.

Functions

- **Syntax :**

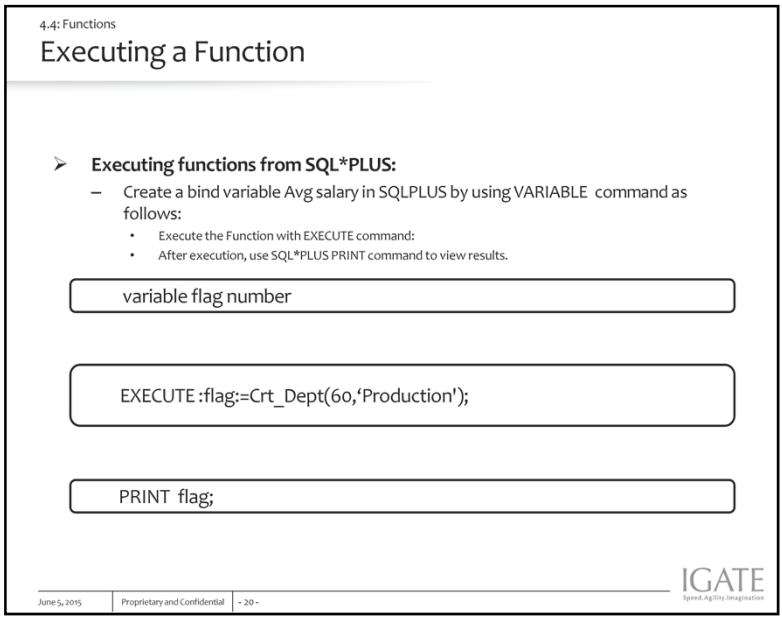
```
CREATE  FUNCTION Func_Name(Param datatype :=
  value,..) RETURN datatype1 AS
        Variable_Declaration ;
        Cursor_Declaration ;
        Exception_Declaration ;
  BEGIN
          PL/SQL_Statements ;
          RETURN Variable_Or_Value_Of_Type_Datatype1 ;
  EXCEPTION
          Exception_Definition ;
  END Func_Name ;
```



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;
```

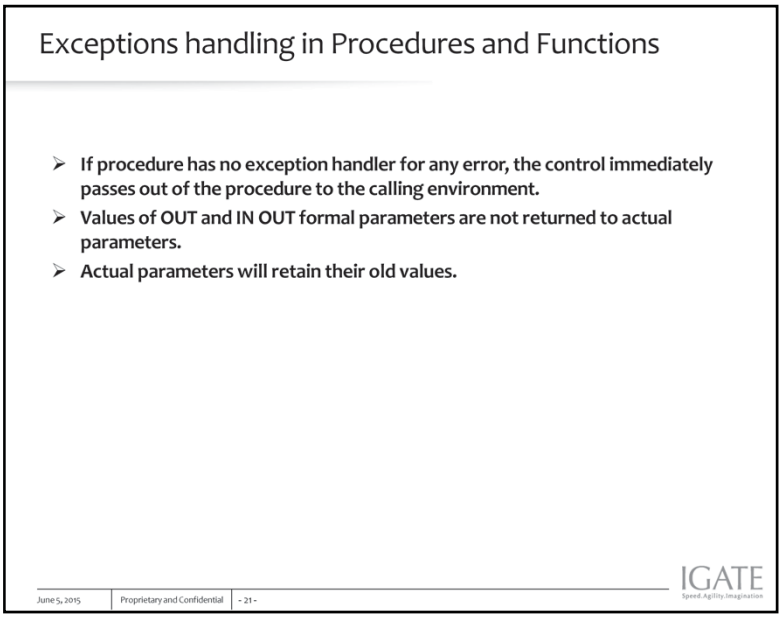


Functions can also be executed through command line as shown on the slide or can be called from other procedures/functions/Anonymous PL/SQL blocks.
The second snippet calls the function and passes the actual parameters. The variable declared earlier is used for collecting the return value from the function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

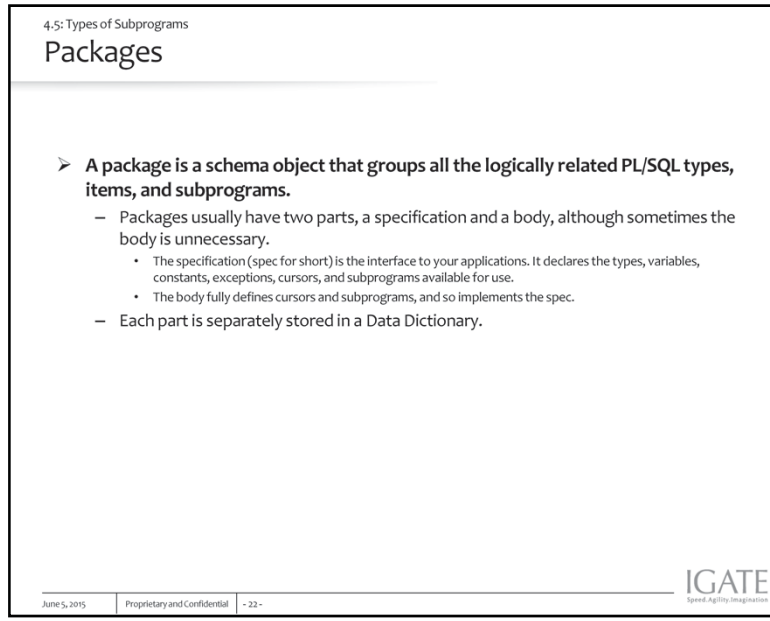
Calling function using a Select statement

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



Exceptions raised inside Procedures and Functions:

- If an error occurs inside a procedure, an exception (pre-defined or user-defined) is raised.

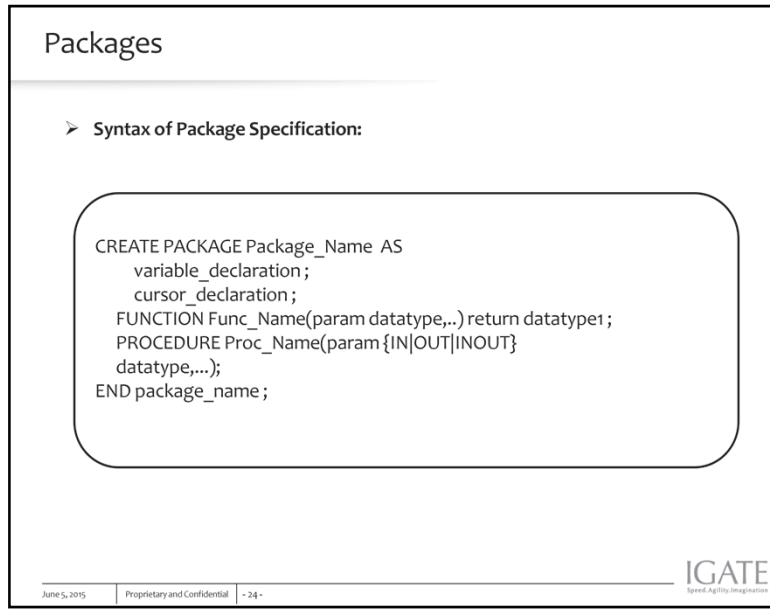


Packages:

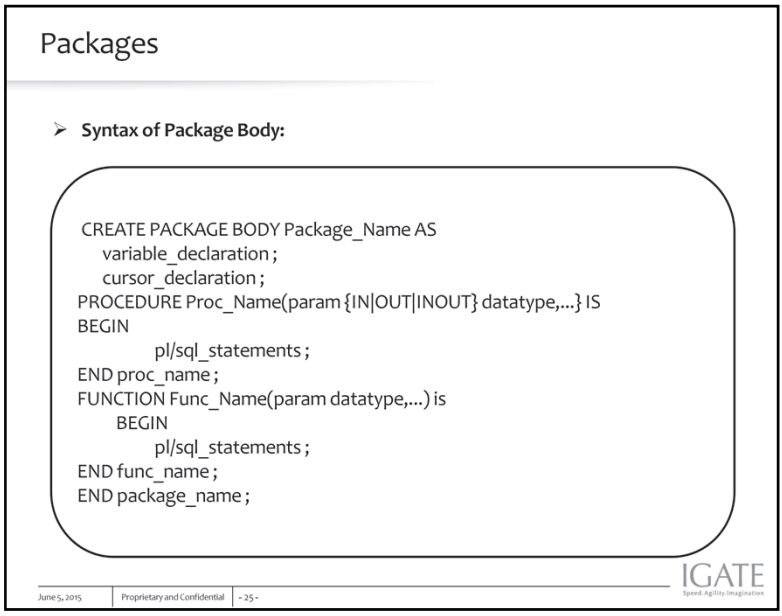
- Packages are PL/SQL constructs that allow related objects to be stored together. A Package consists of two parts, namely "Package Specification" and "Package Body". Each of them is stored separately in a "Data Dictionary".
- **Package Specification:** It is used to declare functions and procedures that are part of the package. Package Specification also contains variable and cursor declarations, which are used by the functions and procedures. Any object declared in a Package Specification can be referenced from other PL/SQL blocks. So Packages provide global variables to PL/SQL.
- **Package Body:** It contains the function and procedure definitions, which are declared in the Package Specification. The Package Body is optional. If the Package Specification does not contain any procedures or functions and contains only variable and cursor declarations then the body need not be present.
- All functions and procedures declared in the Package Specification are accessible to all users who have permissions to access the Package. Users cannot access subprograms, which are defined in the Package Body but not declared in the Package Specification. They can only be accessed by the subprograms within the Package Body. This facility is used to hide unwanted or sensitive information from users.
- A Package generally consists of functions and procedures, which are required by a specific application or a particular module of an application.

Packages

- **Note that:**
  - Packages variables ~ global variables
  - Functions and Procedures ~ accessible to users having access to the package
  - Private Subprograms ~ not accessible to users



The package specification can contain variables,cursors, procedure and functions. Whatever is specified within the packages are global by default and are accessible to users who have the privileges on the package



The package body should contain all the procedures and function declared in the package specification. Any variables and cursors declared within the package body are local to the package body and are accessible only within the package. The package body can contain additional procedures and functions apart from the ones declared in package body. The procedures/functions are local to the package and cannot be accessed by any user outside the package.

4.5: Packages

Example of Package

- **Creating Package Specification**

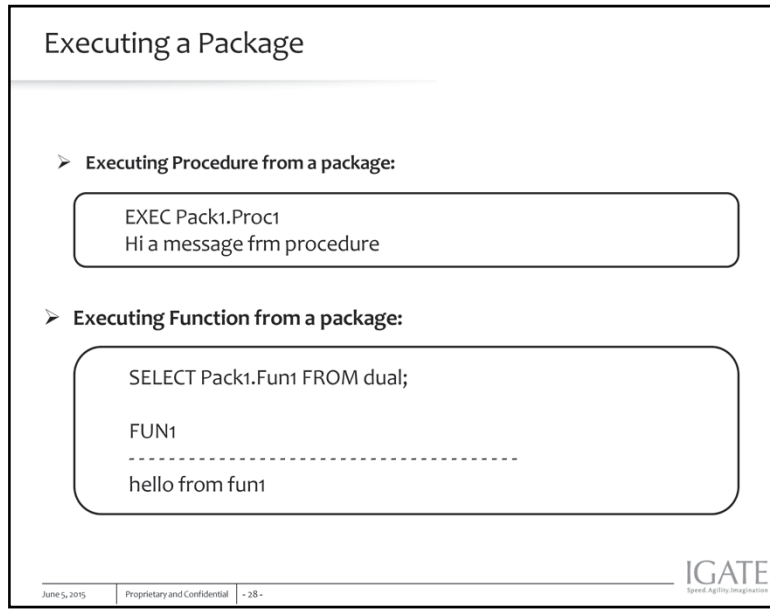
```
CREATE OR REPLACE PACKAGE Pack1 AS
        PROCEDURE Proc1;
        FUNCTION Fun1 return varchar2;
END pack1;
```

4.5: Packages

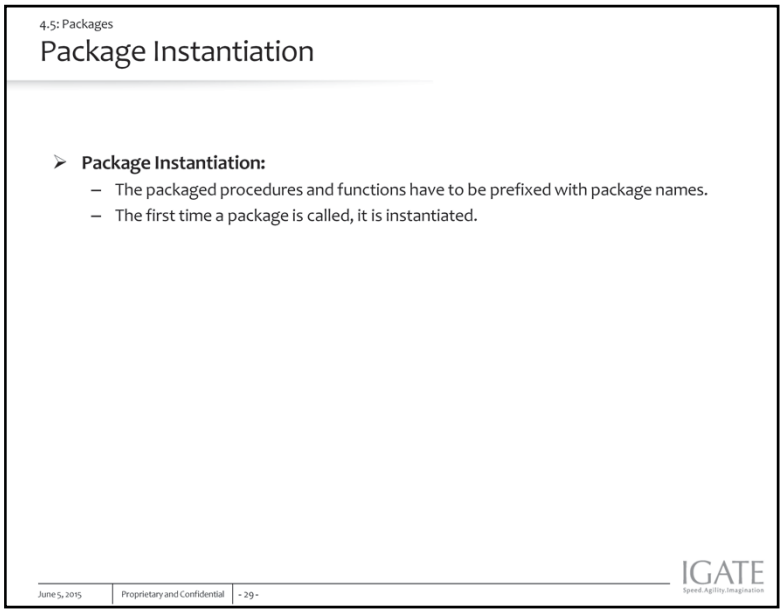
Example of Package

- **Creating Package Body**

```
CREATE OR REPLACE PACKAGE BODY Pack1 AS
  PROCEDURE Proc1 IS
  BEGIN
        dbms_output.put_line('hi a message frm procedure');
  END Proc1;
        function Fun1 return varchar2 IS
  BEGIN
        return ('hello from fun1');
  END Fun1;
END Pack1;
```



Note:
If the specification of the package declares only types, constants, variables, and exceptions, then the package body is not required there. This type of packages only contains global variables that will be used by subprograms or cursors.



Package Instantiation:

- The procedure and function calls are the same as in standalone subprograms.
- The packaged procedures and functions have to be prefixed with package names.
- The first time a package is called, it is instantiated.
  - This means that the package is read from disk into memory, and P-CODE is run.
  - At this point, the memory is allocated for any variables defined in the package.
    - Each session will have its own copy of packaged variables, so there is no problem of two simultaneous sessions accessing the same memory locations.

Subprograms and Ref Type Cursors: Passing a Cursor Variable (contd.):

- In a similar manner, you can pass the Cursor Variable as (: cur) and '2' number for second choice in the EmpData.ret_data procedure. This will give you the output for all the employees who have salary above 2500.
- To see the output of the third cursor, use the same package.procedure name with the ': cur' host variable, and choice value which shows all the employees having department number as 20.
- We can also create a package with the different REF CURSOR TYPES available (that is define the REF CURSOR type in a separate package), and then reference that type in the standalone procedure.
- Example 1: Create a package as shown below:

```
SQL> CREATE or replace PACKAGE Cv_Types AS
                                        TYPE GenericCurTyp IS
REF CURSOR;
TYPE staffCurTyp IS REF CURSOR RETURN
staff_master%ROWTYPE;
              TYPE deptCurTyp IS REF CURSOR RETURN
department_master%ROWTYPE;
 END Cv_Types;
 /
Package created.
```

- Example 2: Create a standalone procedure that references the REF CURSOR type GenericCurTyp, which is defined in the package cv_types. Hence create a procedure as shown below:

```
SQL> CREATE or REPLACE PROCEDURE Open_Pro
(generic_cv IN OUT
      cv_types.GenericCurTyp,choice IN NUMBER) IS
 BEGIN
  IF choice = 1 THEN
OPEN generic_cv FOR SELECT * FROM staff_master;
  ELSIF choice = 2 THEN
       OPEN generic_cv FOR SELECT * FROM
department_master;
  ELSIF choice = 3 THEN
              OPEN generic_cv FOR SELECT * FROM
item;
    END IF;
END Open_Pro;
/
Package created.
```

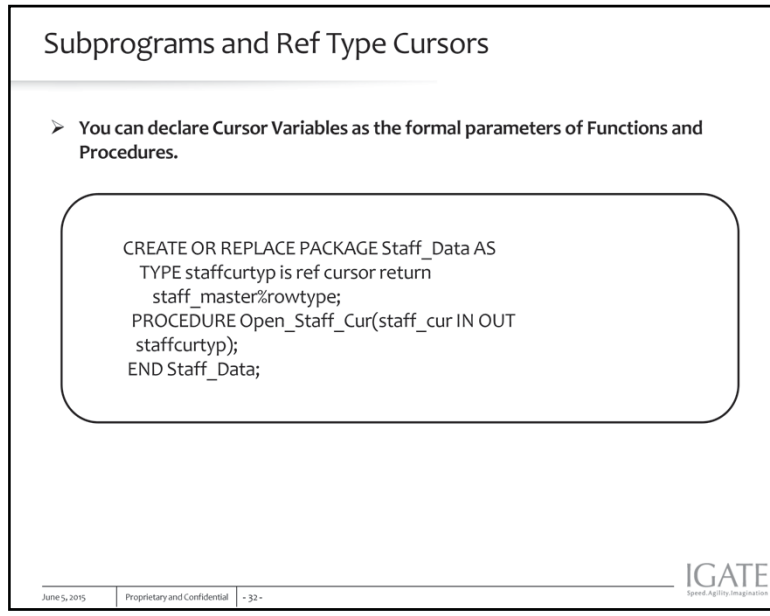
contd.

Subprograms and Ref Type Cursors: Passing a Cursor Variable (contd.):

- Open_ procedure, which has a cursor parameter generic pro is an independent _cv, which refers to type REF CURSOR defined in the cv_types package. You can pass a Cursor Variable and Selector to a stored procedure that executes queries with different return types (that is what you have done in the Open_pro procedure). When you call this procedure with the Generic_cv cursor along with the Selector value, the generic_cv cursor gets open and it retrieves the values from the different tables.
- To execute this procedure you need to create the variable of type REFCURSOR, and pass that variable in the Open_pro procedure to see the output.
- For example:

```
SQL> execute Open_Pro(:cv,2);
```

- This output is that for the choice number 2, that is the Cursor Variable will show all the rows from the Dept table.



Subprograms and Ref Type Cursors:

- You can declare Cursor Variables as the formal parameters of Functions and Procedures.
- In the following example, you define the REF CURSOR type staffCurTyp, then declare a Cursor Variable of that type as the formal parameter of a procedure:

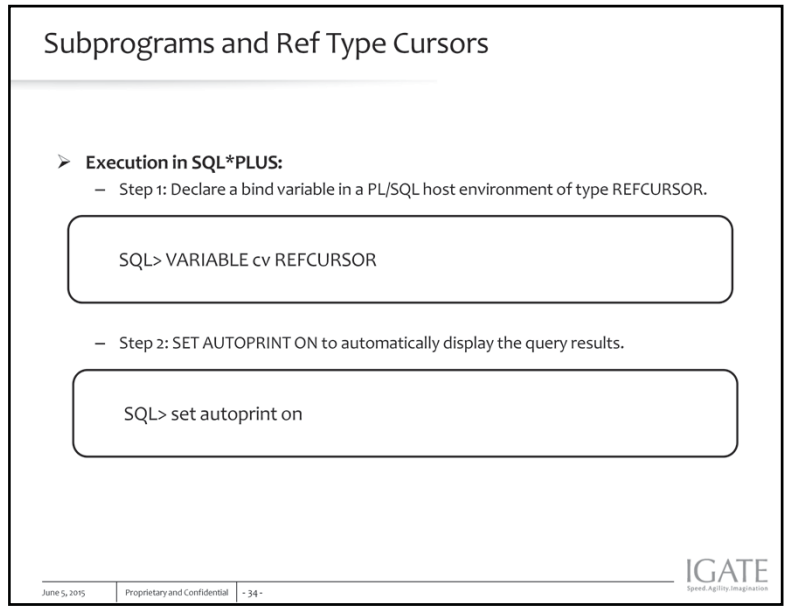
```
DECLARE
TYPE staffCurTyp IS REF CURSOR RETURN
staff_master%ROWTYPE;
PROCEDURE Open_Staff_Cv (staff_cv INOUT
staffCurTyp) IS
```

- Typically, you open a Cursor Variable by passing it to a stored procedure that declares a Cursor Variable as one of its formal parameters.
- The packaged procedure shown in the slide, for example, opens the cursor variable emp_cur.

Subprograms and Ref Type Cursors

```
CREATE OR REPLACE PACKAGE BODY Staff_Data AS
PROCEDURE Open_Staff_Cur (staff_cur IN OUT staffcurtyp) IS
BEGIN
      OPEN staff_cur  for SELECT * FROM staff_master;
      end Open_Staff_Cur;
END Staff_Data;
```

- **Note: Cursor Variable as the formal parameter should be in IN OUT mode.**



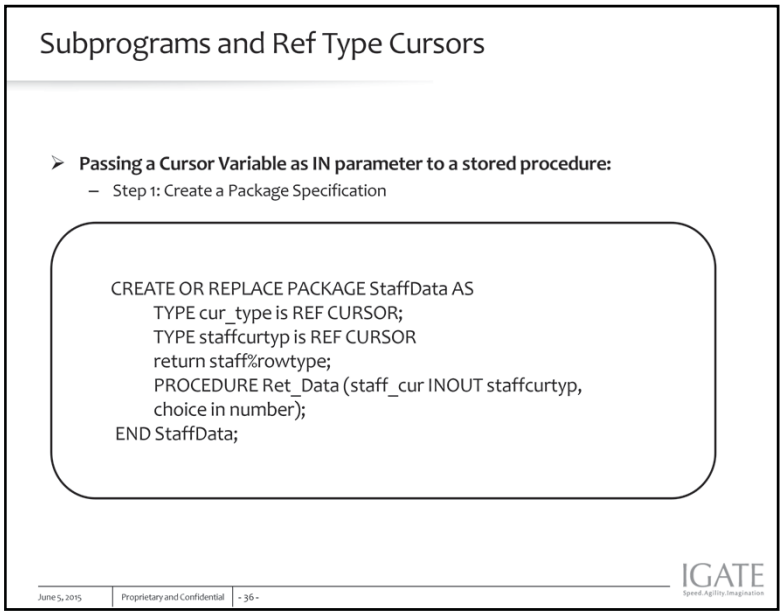
Subprograms and Ref Type Cursors: Execution in SQL*PLUS:

- When you declare a Cursor Variable as the formal parameter of a subprogram that opens the cursor variable, you must specify the IN OUT mode. That way, the subprogram can pass an open cursor back to the caller.
- To see the value of the Cursor Variable on the SQL prompt, you need to do following:
  - Declare a bind variable in a PL/SQL host environment of type REFCURSOR as shown below. The SQL*Plus datatype REFCURSOR lets you declare Cursor Variables, which you can use to return query results from stored subprograms.
    SQL> VARIABLE cv REFCURSOR
  - Use the SQL*Plus command SET AUTOPRINT ON to automatically display the query results.
    SQL> set autoprint on
  - Now execute the package with the specified procedure along with the cursor as follows :
    SQL> execute emp_data.open_emp_cur(:cv);

Subprograms and Ref Type Cursors

- Step 3: Execute the package with the specified procedure along with the cursor as follows:

```
SQL> execute Staff_Data.Open_Staff_Cur(:cv);
```



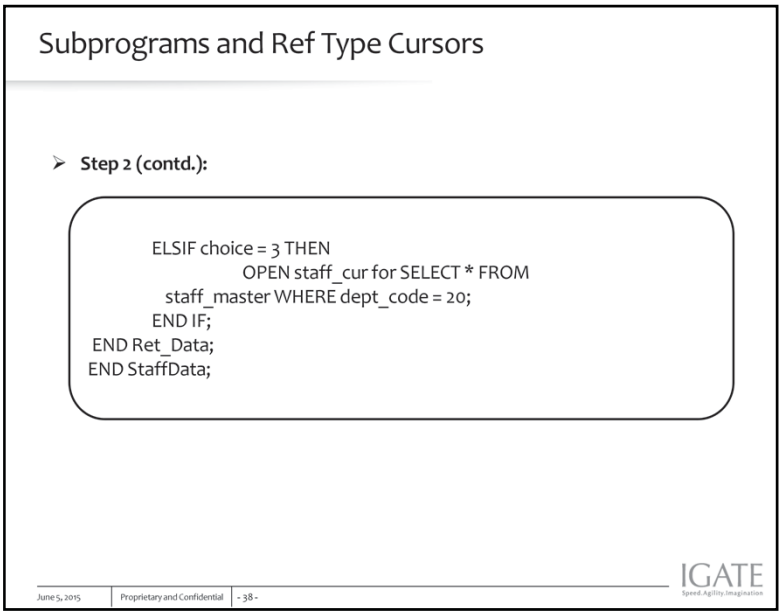
Subprograms and Ref Type Cursors: Passing a Cursor Variable:

- You can pass a Cursor Variable and an IN parameter to a stored procedure, which will execute the queries with different return types.
- In the example shown in the slide, you are passing the cursor as well as the number variable as choice. Depending on the choice you can write multiple queries, and retrieve the output from the cursor.
- When called, the procedure opens the cursor variable emp_cur for the chosen query.

Subprograms and Ref Type Cursors

– Step 2: Create a Package Body:

```
CREATE OR REPLACE PACKAGE BODY StaffData AS
   PROCEDURE Ret_Data (staff_cur INOUT staffcurtyp,
        choice IN number) is
BEGIN
   IF choice = 1 THEN
              OPEN staff_cur for select * FROM staff_master
                WHERE staff_dob is not null;
   ELSIF choice = 2 THEN
               OPEN staff_cur for SELECT * FROM staff_master
            WHERE staff_sal > 2500;
```



Step 3: To retrieve the values from the cursor:

- Define a variable in SQL *PLUS environment using variable command.
- Set the autoprint command on the SQL prompt.
- Call the procedure with the package name and the relevant parameters.

```
SQL> variable cur refcursor
SQL> set autoprint on
SQL> execute StaffData.Ret_Data(:cur,1);
```

4.6 Autonomous Transactions

Autonomous Transactions

- **Autonomous transactions are useful for implementing:**
  - transaction logging,
  - counters, and
  - other such actions, which needs to be performed independent of whether the calling transaction is committed or rolled-back
- **Autonomous transactions:**
  - are independent of the parent transaction.
  - do not inherit the characteristic of the parent (calling) transaction.

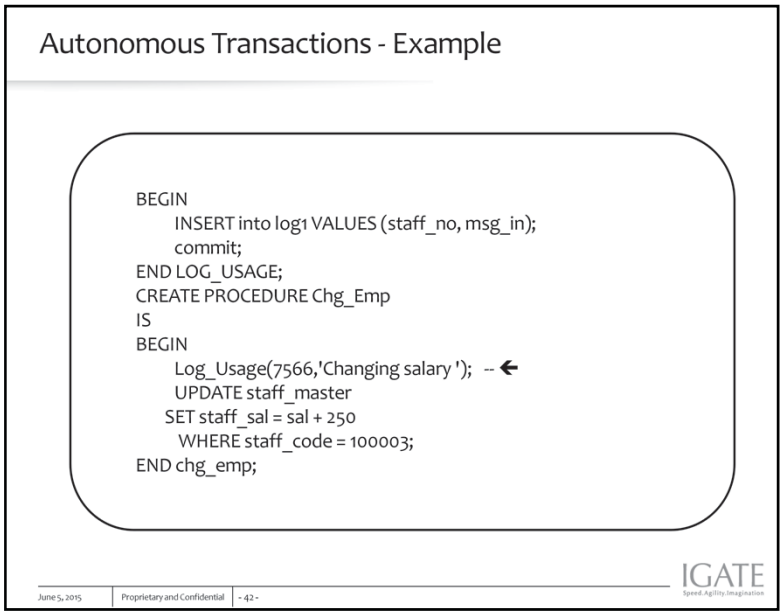
Autonomous Transactions

- **Note that:**
  - Any changes made cannot be seen by the calling transaction unless they are committed.
  - Rollback of the parent does not rollback the called transaction. There are no limits other than the resource limits on how many Autonomous transactions may be nested.
  - Autonomous transactions must be explicitly committed or rolled-back, otherwise an error is generated.

Autonomous Transactions - Example

- **The following example shows how to define an Autonomous block.**

```
CREATE PROCEDURE  Log_Usage ( staff_no IN number,
          msg_in IN varchar2)
IS
PRAGMA AUTONOMOUS_TRANSACTION;
 contd.
```



Note:

- In the example shown in the slide, we are calling log_usage with the employee number and the appropriate message. Then we are updating the corresponding employee record.
- Irrespective of whether the update is successful or not, the insert in the log_usage procedure is always committed.

.

Definer's and Invoker's Rights Model:

- In case of stored procedures, functions and packages (stored subprograms), there are always two situations.
  - First situation is where a stored subprogram is created by a user.
  - Second case is when an already created stored subprogram is invoked by a privileged user of the database.

  By default whenever a subprogram is invoked, it is executed with the privileges of the creating user. This mechanism is called "**definer's rights model**".
- In "definer's rights model", if the stored subprogram (based on EMP table) is created by the user Scott, and another user (say TRG1) executes the stored subprogram, then the privileges of Scott (owner) is used in the context. In this case, even if TRG1 does not have any privileges on the table EMP (owned by Scott), he can still execute the stored subprogram and perform DML operations on the table EMP. This is because the subprogram is executed in the "definer's rights model". This model available as the default model.
- After the Oracle 8i release, the "**invoker's rights model**" can be used. In this model, the procedure executes under the privileges of the user executing the subprogram.
- In the "invokers rights model", if the user Scott creates a stored subprogram, and another user (say TRG1) executes the subprogram, then the privileges of TRG1 (the invoker) will be used in the context of the subprogram rather than the owner of subprogram (Scott).
- In this case, if the user Vivek does not have sufficient rights on the table EMP (owned by Scott), then the invocation of the subprogram will result in an appropriate error message to the invoker.
- Example: As user Scott:

```
CREATE PROCEDURE NAME_COUNT
AUTHID CURRENT_USER
IS
BEGIN
   DECLARE
        N NUMBER;
   BEGIN
   SELECT COUNT(*) INTO N FROM
SCOTT.STAFF_MASTER;
   INSERT INTO STAFFCOUNT VALUES (SYSDATE,
N);
   END;
END;
```

  - Explanation:
    - In line 2, we have defined invoker's rights.
    - In line 8, we are referring to the table EMP from Scott's schema. (This is needed, otherwise Oracle will look for EMP table in the invokers schema)
    - In line 9, the number of employees is inserted into a table empcount which is present in the current users schema.

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

Summary

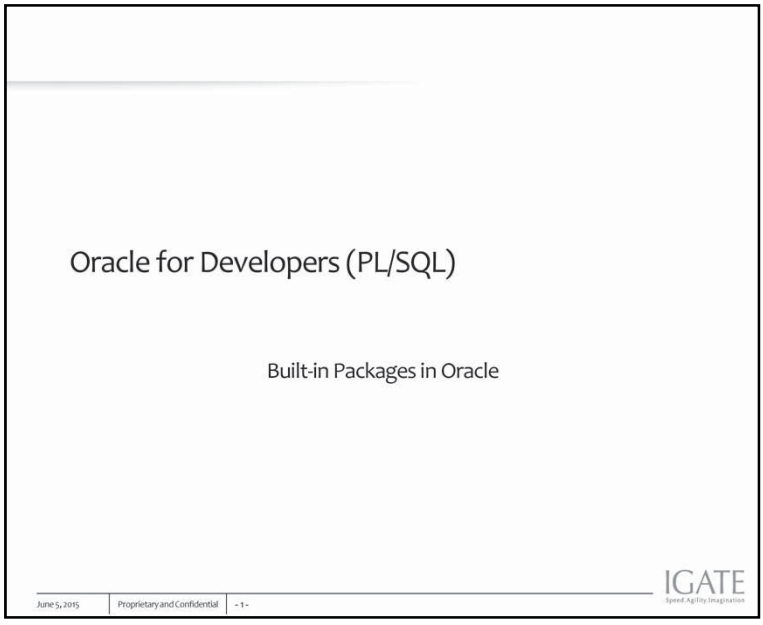
- Functions are used to compute a value
  - A function accepts one or more parameters, and returns a single value by using a return value
  - A function can return multiple values by using OUT parameters
- Packages are schema objects that groups all the logically related PL/SQL types, items, and subprograms
  - Packages usually have two parts, a specification and a body

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
- **Question 3: A Package consists of "Package Specification" and "Package Body", each of them is stored in a Data Dictionary named DBMS_package.**

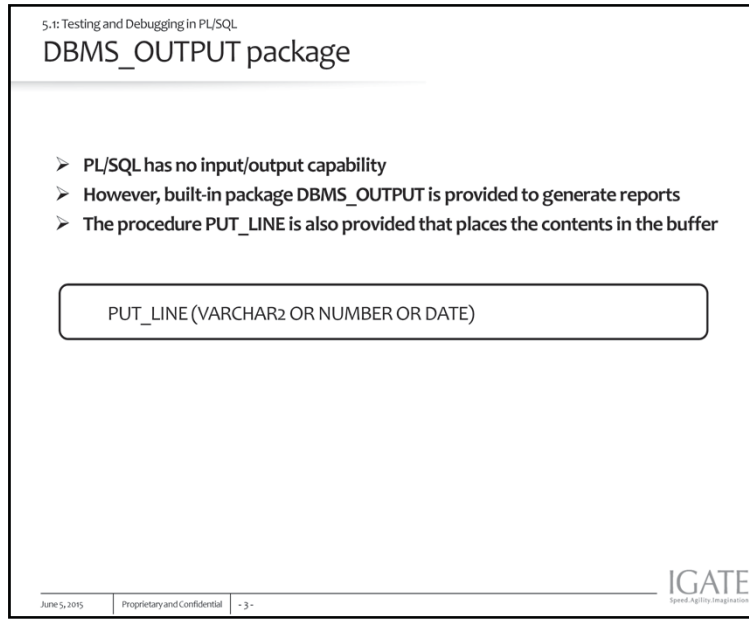
Review – Questions

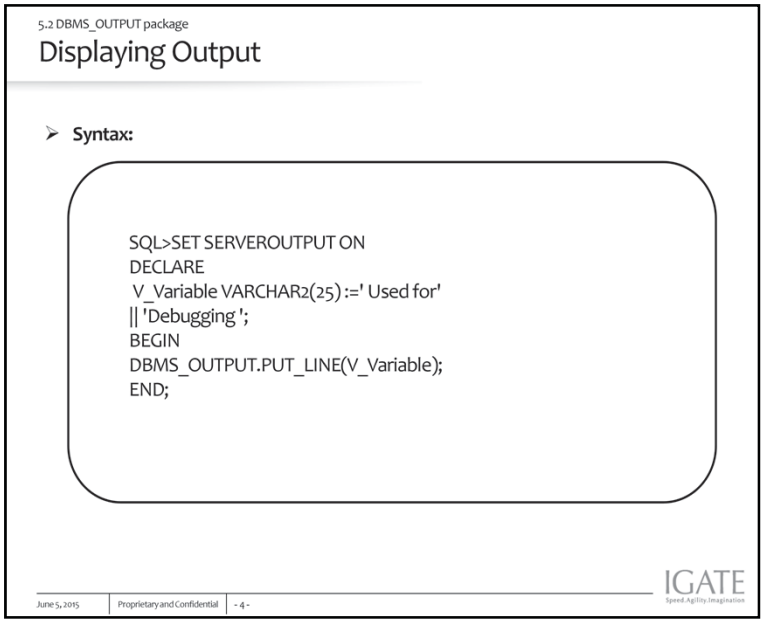
- Question 4: An ___ parameter returns a value to the caller of a subprogram.
- Question 5: A procedure contains two parts: ___ and ___.
- Question 6: In ___ notation, the order of the parameters is not significant.


Oracle for Developers (PL/SQL)
Built-in Packages in Oracle
June 5, 2015
Proprietary and Confidential
- 1 -
IGATE
Speed. Agility. Imagination

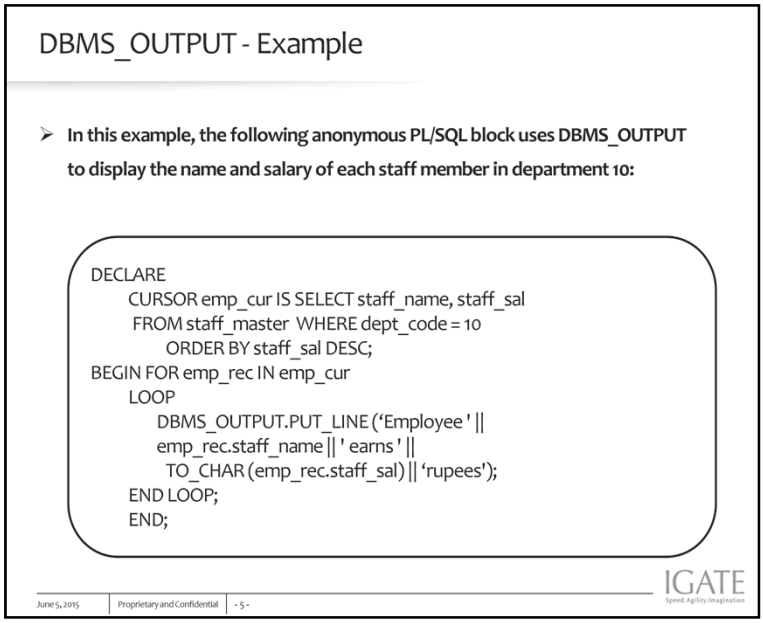
Lesson Objectives

- **To understand the following topics:**
  - Testing and Debugging in PL/SQL
  - DBMS_OUTPUT
  - UTL_file
  - Handling LOB (Large Objects)


5.1: Testing and Debugging in PL/SQL
DBMS_OUTPUT package
PL/SQL has no input/output capability
However, built-in package DBMS_OUTPUT is provided to generate reports
The procedure PUT_LINE is also provided that places the contents in the buffer
PUT_LINE (VARCHAR2 OR NUMBER OR DATE)
June 5, 2015
Proprietary and Confidential
- 3 -
IGATE
Speed.Agility.Imagination



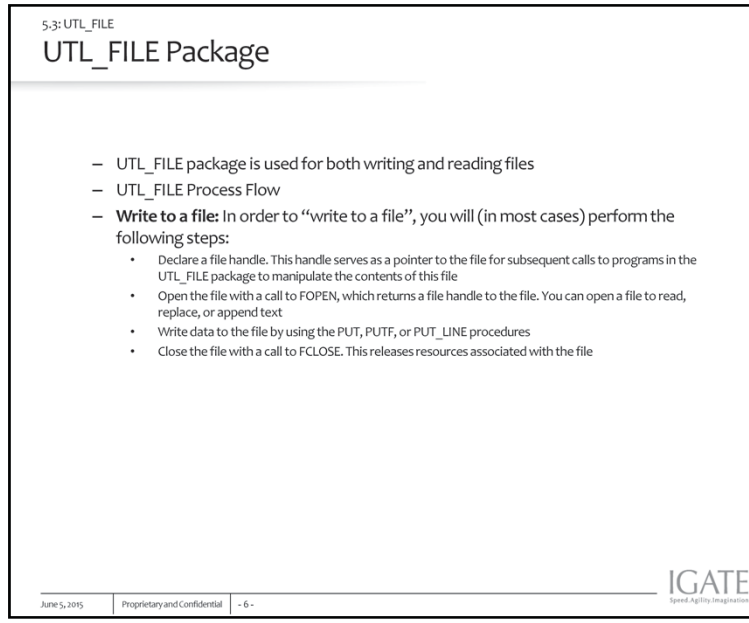
The code will be written on SQL prompt



DBMS_OUTPUT Concepts:

- The program shown in the slide generates the following output when executed in SQL*Plus:

  Employee John earns 32000 rupees
  Employee Mohan earns 24000 rupees.


5.3: UTL_FILE
UTL_FILE Package
– UTL_FILE package is used for both writing and reading files
– UTL_FILE Process Flow
– Write to a file: In order to "write to a file", you will (in most cases) perform the following steps:
• Declare a file handle. This handle serves as a pointer to the file for subsequent calls to programs in the UTL_FILE package to manipulate the contents of this file
• Open the file with a call to FOPEN, which returns a file handle to the file. You can open a file to read, replace, or append text
• Write data to the file by using the PUT, PUTF, or PUT_LINE procedures
• Close the file with a call to FCLOSE. This releases resources associated with the file
June 5, 2015
Proprietary and Confidential
- 6 -
IGATE
Speed.Agility.Imagination

UTL_FILE Process Flow

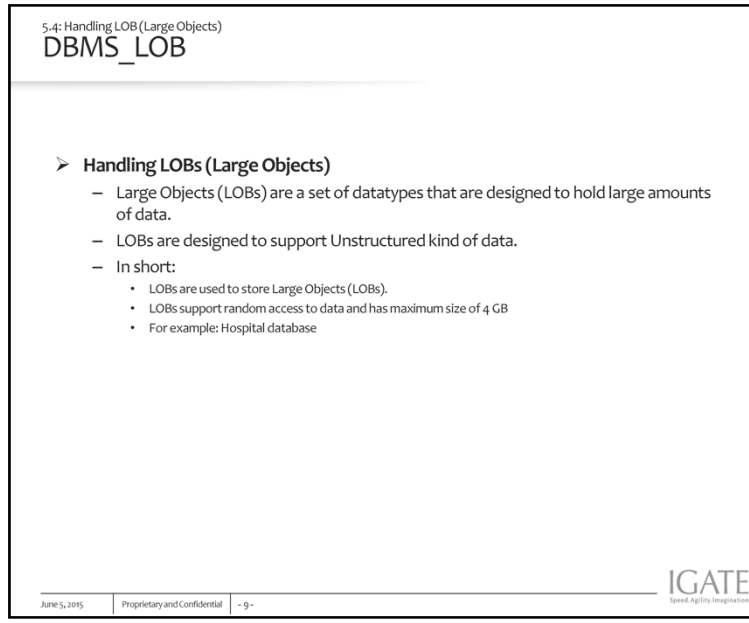
- **Read from a file:** In order to "read data from a file", you will (in most cases) perform the following steps:
  - Declare a file handle
  - Declare a VARCHAR2 string buffer that will receive the line of data from the file. You can also read directly from a file into a numeric or date buffer. In this case, the data in the file will be converted implicitly, and so it must be compatible with the datatype of the buffer
  - Open the file using FOPEN in read mode
  - Use the GET_LINE procedure to read data from the file and into the buffer. To read all the lines from a file, you would execute GET_LINE in a loop
  - Close the file with a call to FCLOSE

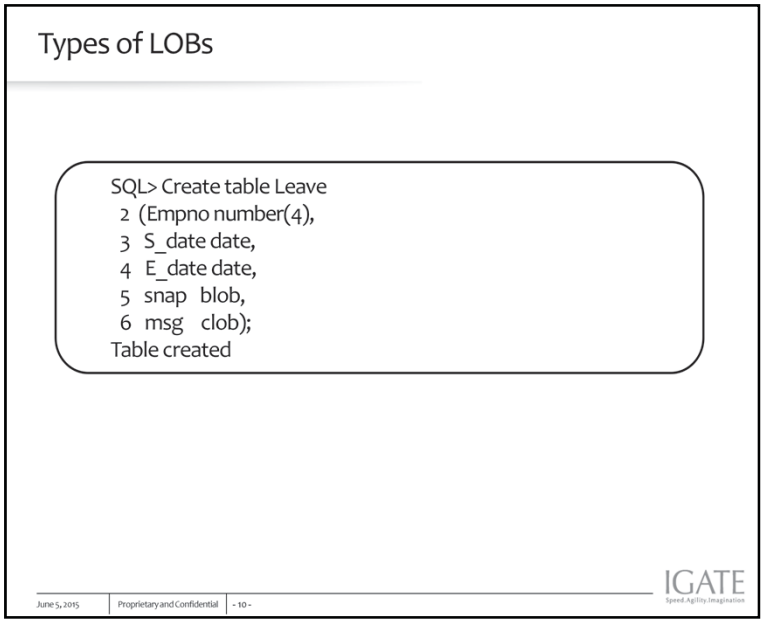
- **The above code will open the file called sample1.txt present on c:\sampledata of the oracle server machine in read mode and will display the first line of the file on the console**
- **Writing to the file requires write permission on the oracle server hard disk which is not given to participants**

```
PROCEDURE FILECLOSE (
  file_loc IN OUT BFILE);
```

Parameter Name	Meaning
File_loc	Locator for the BFILE to be closed.

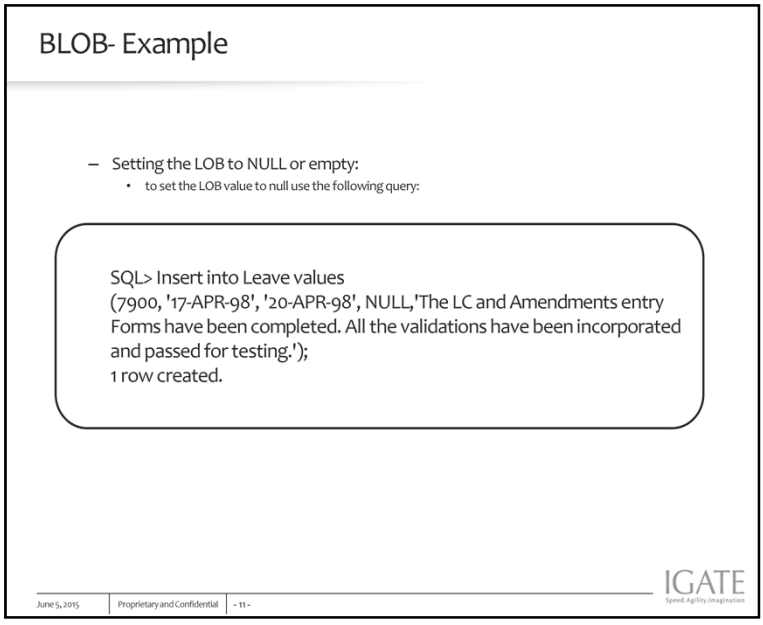
```
PROCEDURE Example_5 IS
  fil BFILE;
BEGIN
  SELECT f_lob INTO fil FROM lob_table WHERE
key_value = 99;
  DBMS_LOB.FILEOPEN(fil);
  -- file operations
  DBMS_LOB.FILECLOSE(fil);
  EXCEPTION
    WHEN some_exception
    THEN handle_exception;
END;
```


5.4: Handling LOB (Large Objects)
DBMS_LOB
Handling LOBs (Large Objects)
Large Objects (LOBs) are a set of datatypes that are designed to hold large amounts of data.
LOBs are designed to support Unstructured kind of data.
In short:
LOBs are used to store Large Objects (LOBs).
LOBs support random access to data and has maximum size of 4 GB
For example: Hospital database
June 5, 2015
Proprietary and Confidential
- 9 -
IGATE
Speed.Agility.Imagination



LOB locator:

- The value held in a LOB column or variable is not the actual binary data, but a "locator" or pointer to the physical location of the large object.
- For internal LOBs, since one LOB value can be up to four gigabytes in size, the binary data will be stored "out of line" (i.e., physically separate) from the other column values of a row (unless otherwise specified; see the next paragraph).
  - This allows the physical size of an individual row to be minimized for improved performance (the LOB column contains only a pointer to the large object).
  - Operations involving multiple rows, such as full table scans, can be more efficiently performed.
- A user can specify that the LOB value be stored in the row itself. This is usually done when working with small LOB values. This approach decreases the time needed to obtain the LOB value. However, the LOB data is migrated out of the row when it gets too big.
- For external LOBs, the BFILE value represents a filename and an operating system directory, which is also a pointer to the location of the large object.

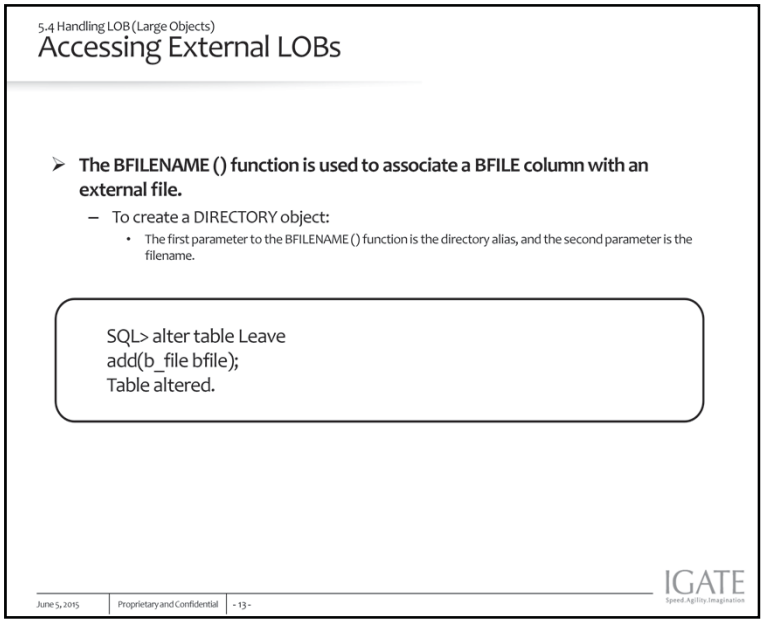


Refer to Appendix for more information on LOBs

BLOB - Example

- Setting the LOB to non-NULL:
  - Before writing data to an internal LOB, column must be made NON-NULL, since you cannot call the OCI or the PL/SQL DBMS_LOB functions on a NULL LOB.

```
SQL> Insert into leave values
 2  (7439,'12-APR-98', '17-APR-98', empty_blob(),
 3  'The assignments regarding Oracle 8 have
 4  been completed. I''ll be back on 17th');
1 row created.
```


5.4 Handling LOB (Large Objects)
Accessing External LOBs
The BFILENAME () function is used to associate a BFILE column with an external file.
To create a DIRECTORY object:
The first parameter to the BFILENAME () function is the directory alias, and the second parameter is the filename.
SQL> alter table Leave
add(b_file bfile);
Table altered.
June 5, 2015
Proprietary and Confidential
- 13 -
IGATE
Speed.Agility.Imagination

BFILE - Example

- **To create a procedure that displays the first 30 characters of the file, refer the following example:**

```
CREATE OR REPLACE PROCEDURE proc_bfile(Eno in number) IS
LOC BFILE;
      V_FILEEXISTS INTEGER;
      v_FILEISOPEN INTEGER;
      NUM NUMBER;
      OFFSET NUMBER;
      LEN NUMBER;
      DIR_ALIAS VARCHAR2(5);
      NAME VARCHAR2(15);
      CONTENTS LONG;
contd.
```

BFILE - Example

```
BEGIN
      SELECT B_FILE INTO LOC FROM LEAVE WHERE EMPNO=Eno;
-- Check to see if file exists
 V_FILEEXISTS := DBMS_LOB.FILEEXISTS(LOC);
 IF v_FILEEXISTS = 1 THEN
      DBMS_OUTPUT.PUT_LINE('The file exists');
 ELSE
 GoTo E;
 END IF;
-- Check if file open
 v_FILEISOPEN := DBMS_LOB.FILEISOPEN( LOC );
contd.
```

BFILE - Example

```
--Determine actions if file is opened or not
 IF v_FILEISOPEN = 1 THEN
       DBMS_OUTPUT.PUT_LINE('The file is open');
 ELSE
       DBMS_OUTPUT.PUT_LINE('Opening the file');
       DBMS_LOB.FILEOPEN(LOC);
       LEN := DBMS_LOB.GETLENGTH( LOC );
       DBMS_OUTPUT.PUT_LINE('Length of the file : ' ||
       TO_CHAR(LEN));
       NUM := 40; OFFSET := 1;
       DBMS_LOB.READ(LOC, NUM, OFFSET, CONTENTS);
contd.
```

BFILE - Example

```
        DBMS_OUTPUT.PUT_LINE('Contents of the file : '||
 CONTENTS);
END IF;
DBMS_LOB.FILEGETNAME(LOC, DIR_ALIAS, NAME);
DBMS_OUTPUT.PUT_LINE ('Opening ' || dir_alias ||
name);
DBMS_LOB.FILECLOSE(LOC); -- Close the BFILE
<<E>>
DBMS_OUTPUT.PUT_LINE('The file cannot be found');
END; /
```

BFILE - Example

- Test the procedure by executing it at SQL prompt as follows:
  - MSG: PL/SQL procedure successfully completed.

```
SQL>  execute proc_bfile(7900);
The file exists
Opening the file
Length of the file : 30
Contents of the file: BOSTONS MANAGEMENT CONSULTANTS
Opening L_DIR TEST.TXT
```

Summary

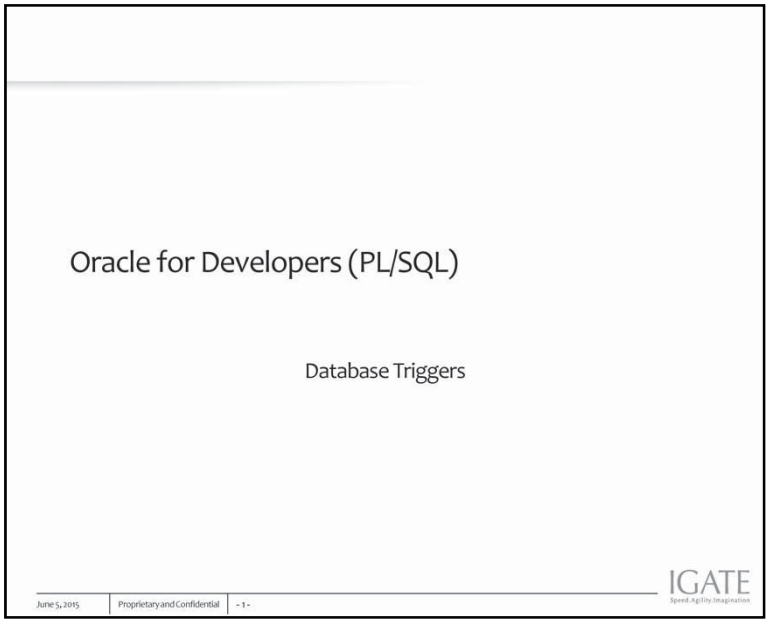
- **In this lesson, you have learnt about:**
  - Testing and Debugging in PL/SQL
  - DBMS_OUTPUT
  - Enabling and Disabling output
  - Writing to the DBMS_OUTPUT Buffer
  - UTL_file
  - Handling LOB (Large Objects)

Review Questions

- **Question 1: The value held in a LOB column or variable is not the actual binary data, but a "locator" or pointer to the physical location of the large object.**
  - True / False
- **Question 2: CLOB stores a column that supports multi-byte characters from National Character set defined by Oracle.**
  - True / False

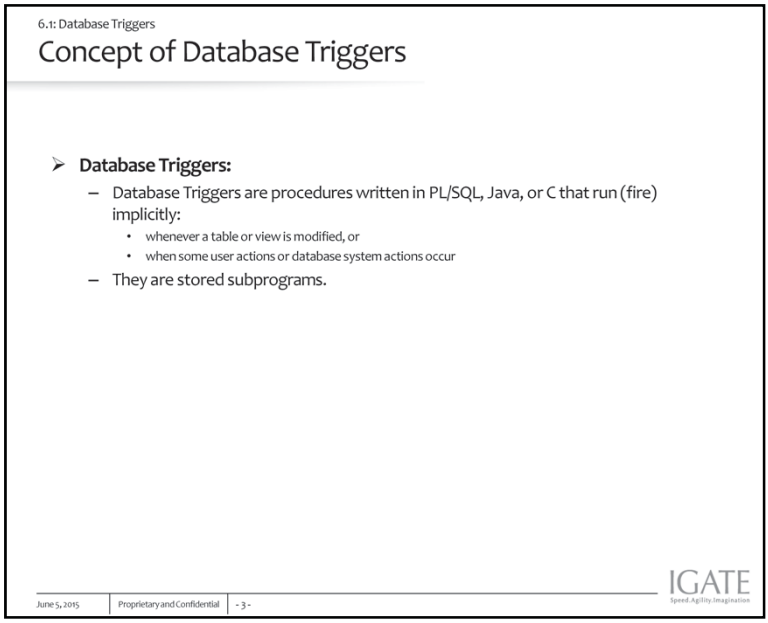
Review Questions

- **Question 3: If the pointer to the file is already located at the last line of the file, UTL_FILE.GET_LINE does not return data.**
  - True / False
- **Question 4: The file can be open in one of the following three modes: ___, ___, and ___.**
- **Question 5: The package ___ lets you read and write files accessible from the server on which your database is running.**


Oracle for Developers (PL/SQL)
Database Triggers
June 5, 2015
Proprietary and Confidential
- 1 -
IGATE
Speed. Agility. Imagination

Lesson Objectives

- **To understand the following topics:**
  - Concept of Database Triggers
  - Types of Triggers
  - Disabling and Dropping Triggers
  - Restriction on Triggers
  - Order of Trigger firing
  - Using :Old and :New values, WHEN clause, Trigger predicates



Database Triggers:

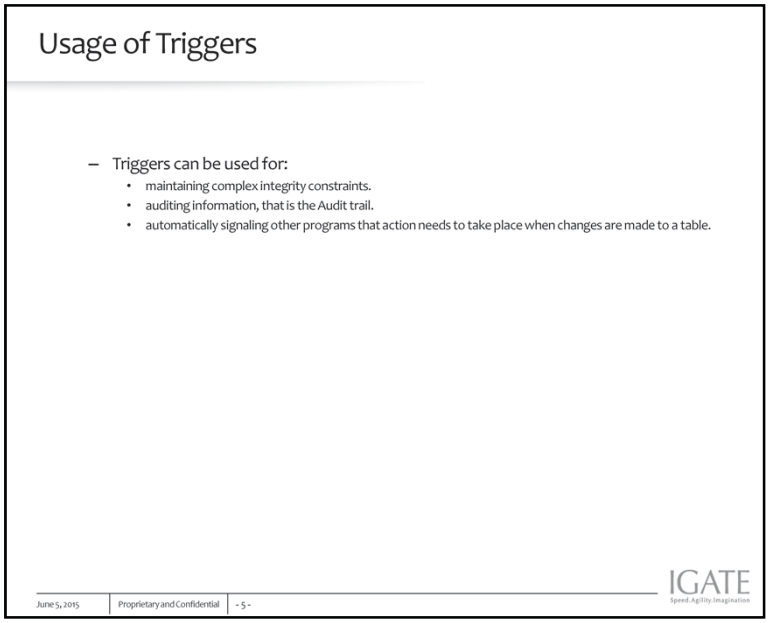
A Trigger defines an action the database should take when some database related event occurs.

Anonymous blocks as well as stored subprograms need to be explicitly invoked by the user.

Database Triggers are PL/SQL blocks, which are implicitly fired by ORACLE RDBMS whenever an event such as INSERT, UPDATE, or DELETE takes place on a table. Database triggers are also stored subprograms.

Concept of Database Triggers

- You can write triggers that fire whenever one of the following operations occur:
  - User events:
    - DML statements on a particular schema object
    - DDL statements issued within a schema or database
    - user logon or logoff events
  - System events:
    - server errors
    - database startup
    - instance shutdown

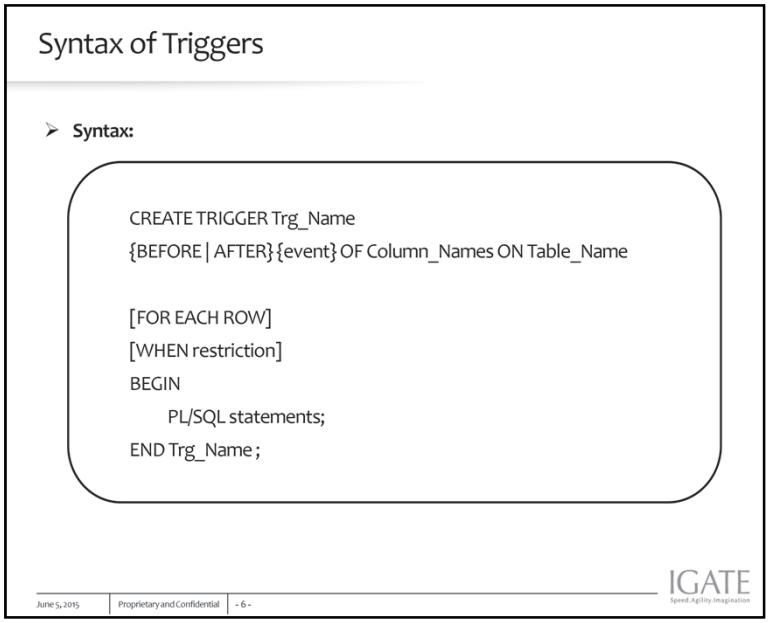


Database Triggers (contd.):

Triggers can be used for:

- Maintaining complex integrity constraints. This is not possible through declarative constraints that are enabled at table creation.
- Auditing information in a table by recording the changes and the identify of the person who made them. This is called as an audit trail.
- Automatically signaling other programs that action needs to take place when changes are made to a table.

contd.



Database Triggers (contd.):

To create a trigger on a table you must be able to alter that table. The slide shows the syntax for creating a table.

contd.

Database Triggers (contd.):

Parts of a Trigger

A trigger has three basic parts:

- A triggering event or statement
- A trigger restriction
- A trigger action

Triggering Event or Statement

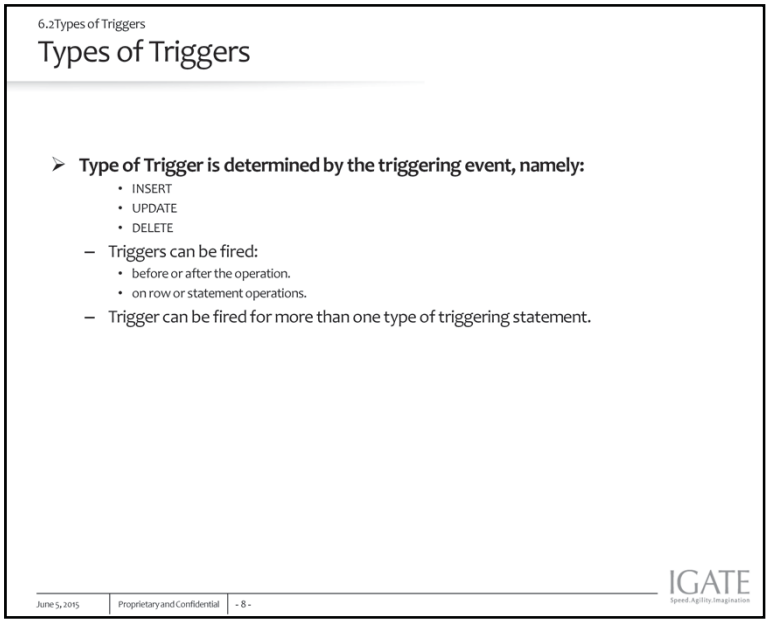
- A triggering event or statement is the SQL statement, database event, or user event that causes a trigger to fire. A triggering event can be one or more of the following:
    - An INSERT, UPDATE, or DELETE statement on a specific table (or view, in some cases)
    - A CREATE, ALTER, or DROP statement on any schema object
    - A database startup or instance shutdown
    - A specific error message or any error message
    - A user logon or logoff

Trigger Restriction

- A trigger restriction specifies a Boolean expression that must be true for the trigger to fire. The trigger action is not run if the trigger restriction evaluates to false or unknown.

Trigger Action

- A trigger action is the procedure (PL/SQL block, Java program, or C callout) that contains the SQL statements and code to be run when the following events occur:
    - a triggering statement is issued
    - the trigger restriction evaluates to true
- Like stored procedures, a trigger action can:
    - contain SQL, PL/SQL, or Java statements
    - define PL/SQL language constructs such as variables, constants, cursors, exceptions
    - define Java language constructs
    - call stored procedures


6.2Types of Triggers
Types of Triggers
Type of Trigger is determined by the triggering event, namely:
INSERT
UPDATE
DELETE
Triggers can be fired:
before or after the operation.
on row or statement operations.
Trigger can be fired for more than one type of triggering statement.
June 5, 2015
Proprietary and Confidential
- 8 -
IGATE
Speed.Agility.Imagination

More on Triggers

Category	Values	Comments
Statement	INSERT, DELETE, UPDATE	Defines which kind of DML statement causes the trigger to fire.
Timing	BEFORE, AFTER	Defines whether the trigger fires before the statement is executed or after the statement is executed.
Level	Row or Statement	• If the trigger is a row-level trigger, it fires once for each row affected by the triggering statement. • If the trigger is a statement-level trigger, it fires once, either before or after the statement. • A row-level trigger is identified by the FOR EACH ROW clause in the trigger definition.

Note:

- A trigger can be fired for more than one type of triggering statement. In case of multiple events, OR separates the events.
- From ORACLE 7.1 there is no limit on number of triggers you can write for a table.
    - Earlier you could write maximum of 12 triggers per table, one of each type.

Types of Triggers (contd.)

Row Triggers and Statement Triggers

- When you define a trigger, you can specify the number of times the trigger action has to be run:
  - Once for every row affected by the triggering statement, such as a trigger fired by an UPDATE statement that updates many rows.
  - Once for the triggering statement, no matter how many rows it affects.

Row Triggers

- A Row Trigger is fired each time the table is affected by the triggering statement. For example: If an UPDATE statement updates multiple rows of a table, a row trigger is fired once for each row affected by the UPDATE statement.
- If a triggering statement affects no rows, a row trigger is not run.
- Row triggers are useful if the code in the trigger action depends on data provided by the triggering.

INSTEAD OF Triggers

- INSTEAD OF triggers provide a transparent way of modifying views that cannot be directly modified through DML statements (INSERT, UPDATE, and DELETE). These triggers are called INSTEAD OF triggers because, unlike other types of triggers, Oracle fires the trigger instead of executing the triggering statement.
- You can write normal INSERT, UPDATE, and DELETE statements against the view and the INSTEAD OF trigger is fired to update the underlying tables appropriately. INSTEAD OF triggers are activated for each row of the view that gets modified.

Modify Views

- Modifying views can have ambiguous results:
  - Deleting a row in a view could either mean deleting it from the base table or updating some values so that it is no longer selected by the view.
  - Inserting a row in a view could either mean inserting a new row into the base table or updating an existing row so that it is projected by the view.
  - Updating a column in a view that involves joins might change the semantics of other columns that are not projected by the view.
- Object views present additional problems. For example, a key use of object views is to represent master/detail relationships. This operation inevitably involves joins, but modifying joins is inherently ambiguous.
- As a result of these ambiguities, there are many restrictions on which views are modifiable. An INSTEAD OF trigger can be used on object views as well as relational views that are not otherwise modifiable.

contd.

Types of Triggers (contd.)

Modify Views (contd.)

- A view is inherently modifiable if data can be inserted, updated, or deleted without using INSTEAD OF triggers and if it conforms to the restrictions. Even if the view is inherently modifiable, you might want to perform validations on the values being inserted, updated or deleted. INSTEAD OF triggers can also be used in this case. Here the trigger code performs the validation on the rows being modified and if valid, propagates the changes to the underlying tables, statement or rows that are affected.

Statement Triggers

- A statement trigger is fired once on behalf of the triggering statement, regardless of the number of rows in the table that the triggering statement affects, even if no rows are affected. For example: If a DELETE statement deletes several rows from a table, a statement-level DELETE trigger is fired only once.
- Statement triggers are useful if the code in the trigger action does not depend on the data provided by the triggering statement or the rows affected.
- For example: Use a statement trigger to:
    - make a complex security check on the current time or user
    - generate a single audit record

BEFORE and AFTER Triggers

- When defining a trigger, you can specify the trigger timing — whether the trigger action has to be run before or after the triggering statement.
- BEFORE and AFTER apply to both statement and row triggers.
- BEFORE and AFTER triggers fired by DML statements can be defined only on tables, not on views. However, triggers on the base tables of a view are fired if an INSERT, UPDATE, or DELETE statement is issued against the view. BEFORE and AFTER triggers fired by DDL statements can be defined only on the database or a schema, and not on particular tables.

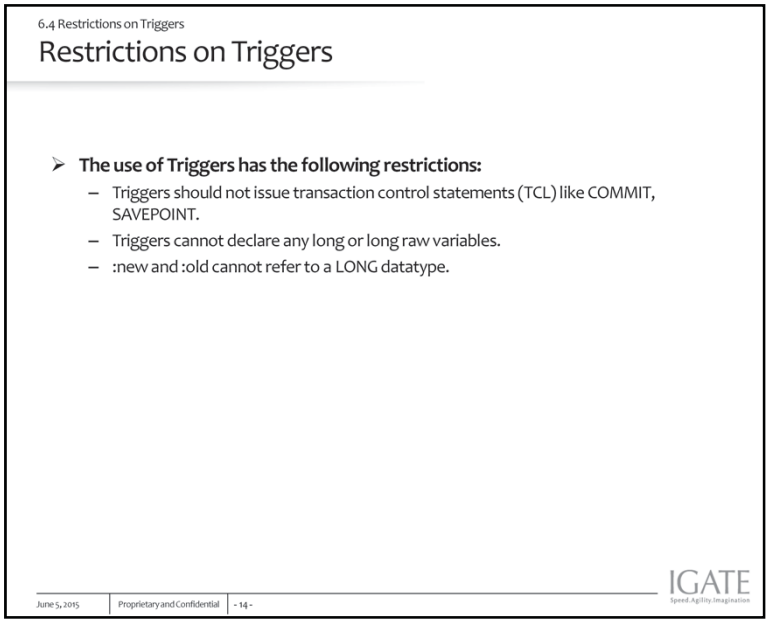
Examples on Trigger

Example 1

```
CREATE TABLE Account_log
(
deleteInfo VARCHAR2(20),
logging_date DATE
)
```

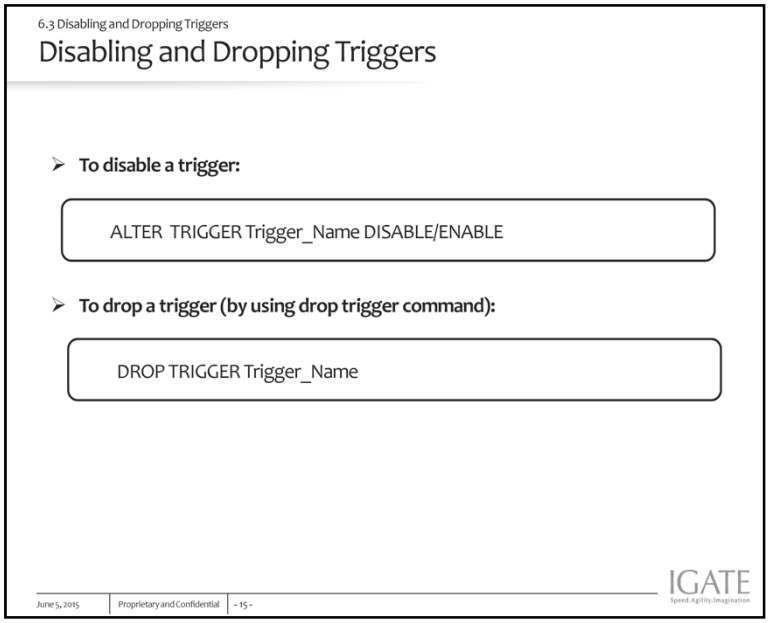
Trigger creation code

```
CREATE or REPLACE TRIGGER

After_Delete_Row_product

 AFTER   delete On Account_masters
 FOR EACH ROW
 BEGIN
 INSERT INTO  Account_log
 Values('After delete, Row level',sysdate);
 END;
```



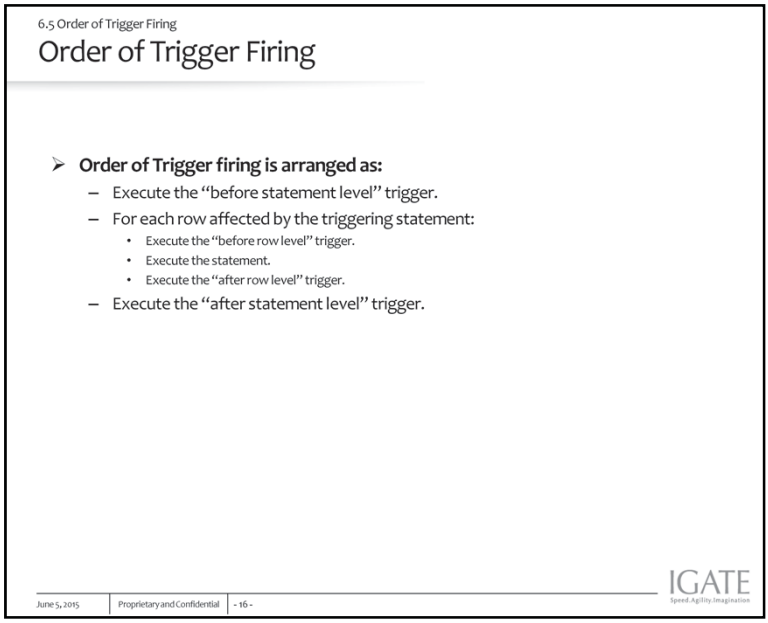
Restrictions on Triggers:

- A Trigger may not issue any transaction control statements (TCL) like COMMIT, SAVEPOINT.
    - Since the triggering statement and the trigger are part of the same transaction, whenever triggering statement is committed or rolled back the work done in the trigger gets committed or rolled back, as well.
- Any procedures or functions that are called by the Trigger cannot have any transaction control statements.
- Trigger body cannot declare any long or long raw variables. Also :new and :old cannot refer to a LONG datatype.
- There are restrictions on the tables that a trigger body can access depending on type of "triggers" and "constraints" on the table.



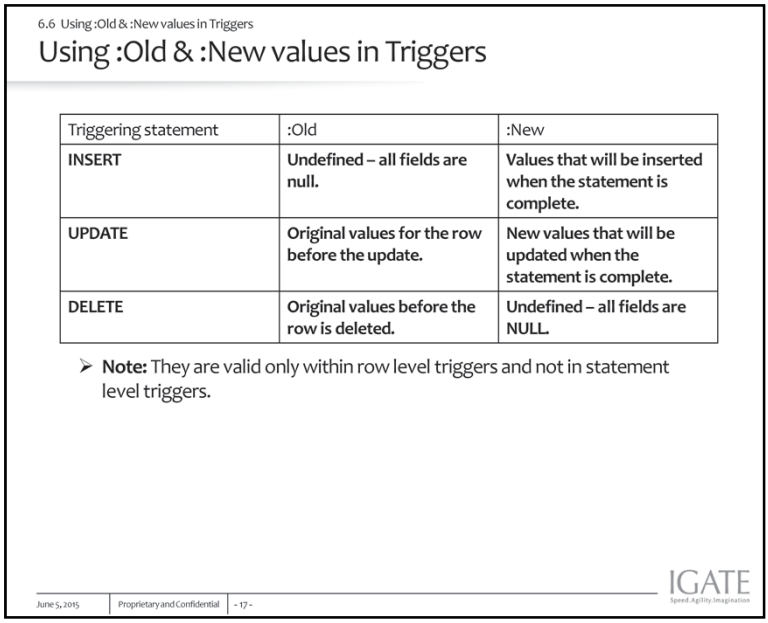
Note:

An UPDATE or DELETE statement affects multiple rows of a table.
If the trigger is fired once for each affected row, then FOR EACH ROW clause is required. Such a trigger is called a ROW trigger.
If the FOR EACH ROW clause is absent, then the trigger is fired only once for the entire statement. Such triggers are called STATEMENT triggers



Note:

- As each trigger is fired, it will see the changes made by the earlier triggers, as well as the database changes made by the statement.
- The order in which triggers of same type are fired is not defined.
- If the order is important, combine all the operations into one trigger.

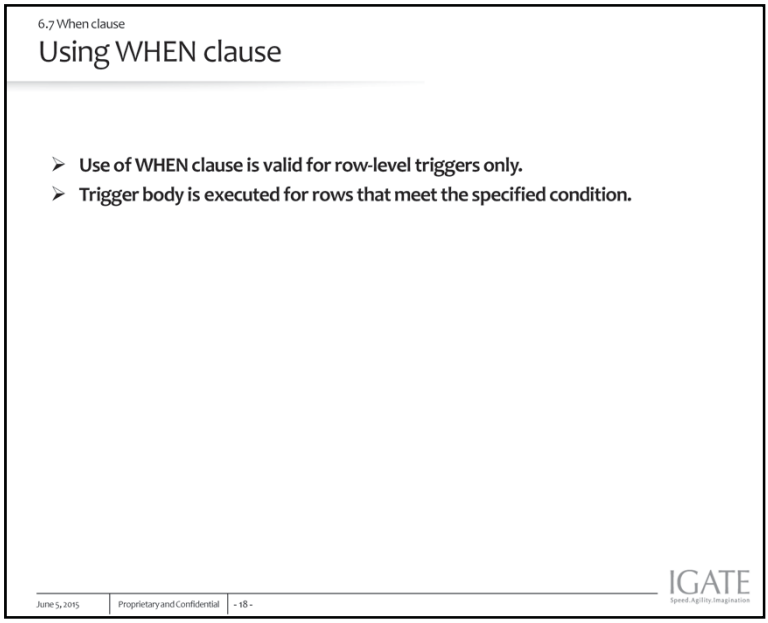


Using :old and :new values in Row Level Triggers:

- Row level trigger fires once per row that is being processed by the triggering statement.
- Inside the trigger, you can access the row that is currently being processed. This is done through keywords :new and :old (they are called as pseudo records). The meaning of the terms is as shown in the slide.

Note:

- The pseudo records are valid only within row level triggers and not in statement level triggers.
  - :old values are not available if the triggering statement is INSERT.
  - :new values are not available if the triggering statement is DELETE.
- Each column is referenced by using the notation :old.Column_Name or :new.Column_Name.
- If a column is not updated by the triggering update statement, then :old and :new values remain the same.



Using WHEN Clause:

- The WHEN clause is valid for row-level triggers only. If present, the trigger body will be executed only for those rows that meet the condition specified by the WHEN clause.
- The :new and :old records can be used here without colon.

contd.

6.8 Examples on Trigger

Example 2

```
CREATE TABLE Account_masters
(
account_no NUMBER(6) PRIMARY KEY,
cust_id NUMBER(6),
account_type CHAR(3) CONSTRAINT chk_acc_type CHECK(account_type IN
('SAV','SAL')),
Ledger_balance NUMBER(10)
)
```

Slide 19

TKA1 THIS EXAMPLE IS NEWLY ADDED
and it is based on

CREATE OR REPLACE TRIGGER trg_acc_master_ledger
before INSERT OR UPDATE FOR EACH ROW
Tanmaya K Acharya, 1/30/2015

Trigger creation code

```
CREATE OR REPLACE TRIGGER trg_acc_master_ledger
  before INSERT OR UPDATE OF Ledger_balance ,account_type  ON
Account_masters
  FOR EACH ROW

  WHEN (NEW.account_type ='SAV')
  DECLARE

   v_led_bal NUMBER(10);
  BEGIN
   v_led_bal:= :NEW.Ledger_balance;

   IF v_led_bal < 5000 THEN
    :NEW.Ledger_balance := 5000;
   END if;
  END trg_acc_master_ledger;
```

Summary

- **Database Triggers are procedures written in PL/SQL, Java, or C that run (fire) implicitly: Database Triggers are procedures written in PL/SQL, Java, or C that run (fire) implicitly:**
  - whenever a table or view is modified, or
  - when some user actions or database system actions occur
- **There are three types of triggers:**
  - Statement based triggers
  - Timing based triggers
  - Level based triggers

Summary

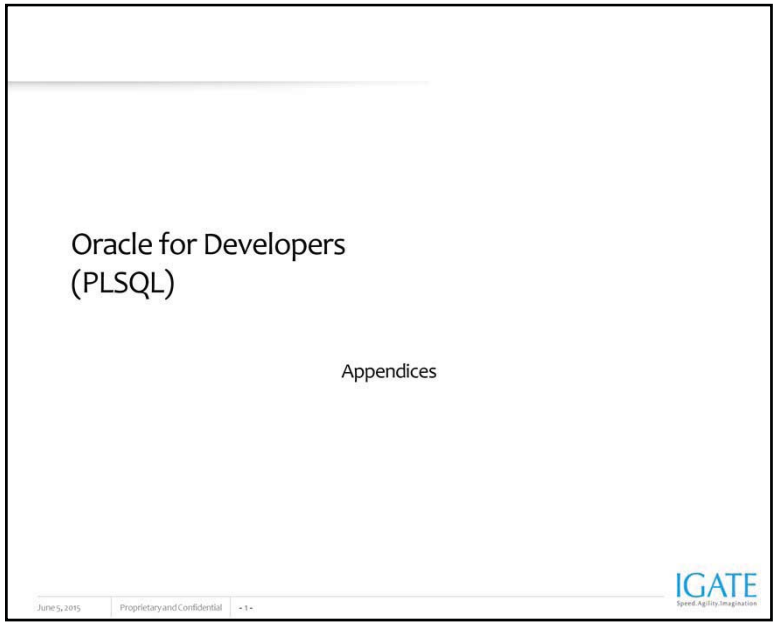
- Disabling and Dropping triggers can be done instead of actually removing the triggers
- Order of trigger firing is decided depending on the type of triggers used in the sequence

Review – Questions

- **Question 1: Triggers should not issue Transaction Control Statements (TCL).**
  - True / False
- **Question 2: BEFORE DROP and AFTER DROP triggers are fired when a schema object is dropped.**
  - True / False
- **Question 3: The :new and :old records must be used in WHEN clause with a colon.**

Review – Questions

- Question 4: A ___ is a table that is currently being modified by a DML statement.
- Question 5: A ___ is fired once on behalf of the triggering statement, regardless of the number of rows in the table that the triggering statement affects.


Oracle for Developers
(PLSQL)
Appendices
June 5, 2015
Proprietary and Confidential
- 1 -
IGATE
Speed. Agility. Imagination

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.

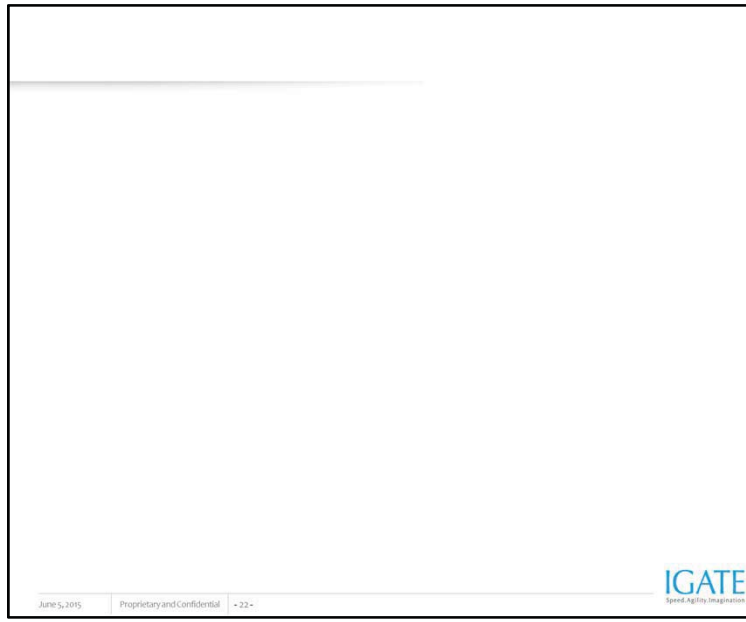
UTL_FILE.FCLOSE procedure:

- 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;
```

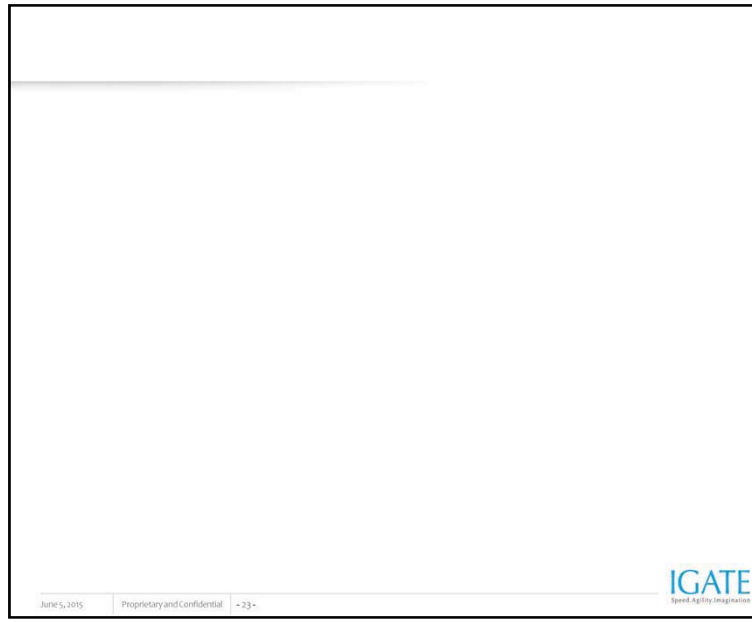
- This procedure will come in handy when you have opened a variety of files and want to make sure that none of them are left open when your program terminates.
- In programs in which files have been opened, you should also call FCLOSE_ALL in exception handlers in programs. If there is an abnormal termination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

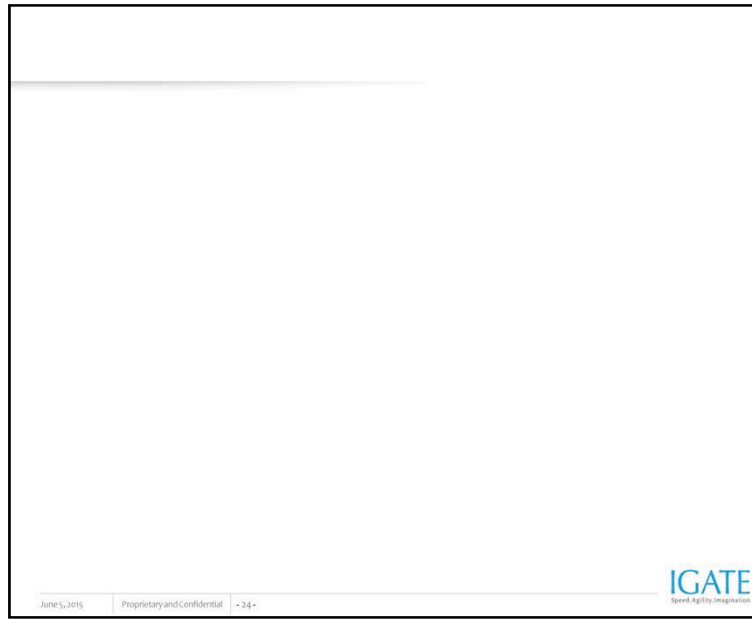
  Those stored as operating system files, such as BFILEs

- LOBs are designed to support Unstructured kind of data.

Unstructured Data:

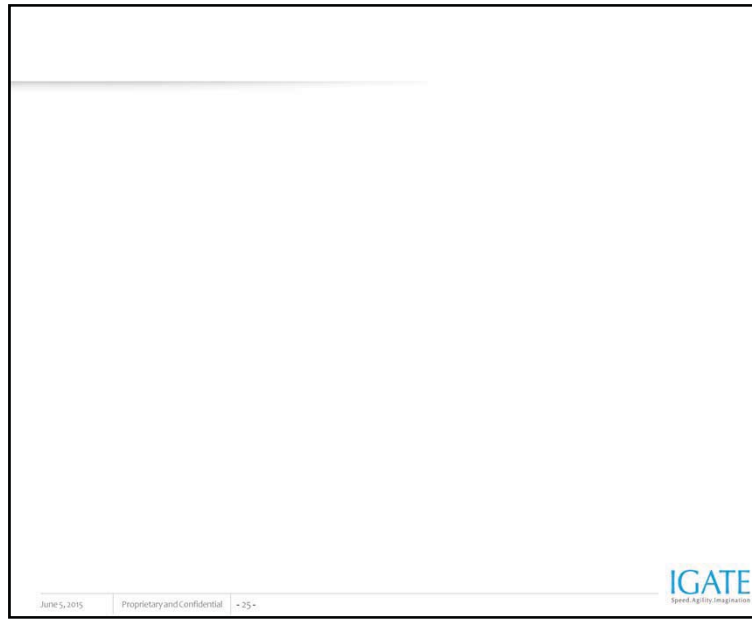
- Unstructured Data cannot be decomposed into Standard Components:
  - Unstructured data cannot be decomposed into standard components. Data about an Employee can be "structured" into a Name (probably a character string), an identification (likely a number), a Salary, and so on. But if you are given 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.



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
- 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.



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:

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

  - 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.

```
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.):

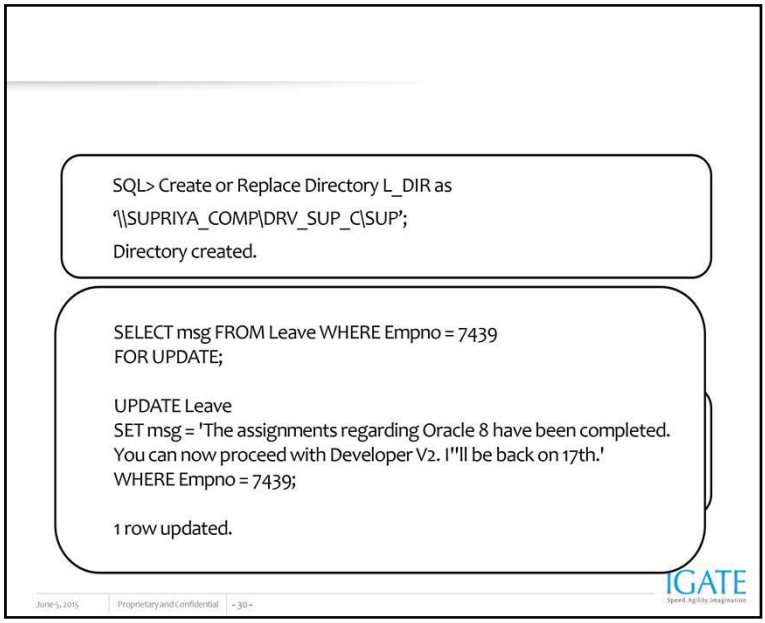
- **NULL versus "empty" LOB locators**
  - Oracle8 provides the built-in functions EMPTY_BLOB and EMPTY_CLOB to set BLOB, CLOB, and NCLOB columns to "empty".
  - For example:

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

  - 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.

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.


IGATE
Speed. Agility. Imagination
June 5, 2015
Proprietary and Confidential
- 33 -

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

Table of Contents

Document Revision History 2
Table of Contents 3
Getting Started 4
Overview 4
Setup Checklist for Oracle 9i 4
Instructions 4
Learning More (Bibliography if applicable) 4
Lab 1. Introduction to PL/SQL and Cursors 5
Lab 2. Exception Handling and Dynamic SQL 7
Lab 3. Database Programming 9
Lab 4. Case Study 1 13
Lab 5 : Case Study 2 15
Lab 6 :Handling Files, DBMS_LOB 17
Appendices 18
Appendix A: Oracle Standards 18
Appendix B: Coding Best Practices 19
Appendix C: Table of Examples 20

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 • PL/SQL variables and data types • Create, Compile and Run anonymous PL/SQL blocks • Usage of Cursors
Time	1hr 30 min

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 to display the details of the employee number 7369. (To Do)

1.4 Write a PL/SQL program to display the details of Department No 10 from dept table. (To Do)

1.5 Write a PL/SQL program to accept the Employee Name and display the details of that Employee including the Department Name. (To Do)

1.6. Write a PL/SQL block to retrieve all staff (staffcode, name, salary) under specific department number and display the result. (Note: The Department_Code will be accepted from user. Cursor needto be used.)

1.7.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

1.8. Write a PL/SQL block to generate the following report for a given Department code

Student_CodeSudent_Name Subject1 Subject2 Subject3 Total Percentage Grade

Note: Display suitable error massage if wrong department code has entered and if there is no student in the given department.

For Grade:
Student should pass in each subject individually (pass marks 60).
Percent >= 80 then grade= A
Percent >= 70 and < 80 then grade= B
Percent >= 60 and < 70 then grade= C
Else D

1.9. Write a PL/SQL block to retrieve the details of the staff belonging to a particular department. Department code should be passed as a parameter to the cursor.

Lab 2.Exception Handling and Dynamic SQL

Goals	Implementing Exception Handling ,Analyzing and Debugging
Time	2hr

2.1: Modify the programs created in Lab2 to implement Exception Handling

2.2 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

2.3 Rewrite the above block to achieve the requirement.
2.4
Predict the output of the following block ?What corrections would be needed to make it more efficient?

```
BEGIN
    DECLARE
fnameemp.ename%TYPE;
    BEGIN
    SELECTenameINTOfname
    FROMemp
    WHERE 1=2;

DBMS_OUTPUT.PUT_LINE('This statement will print');
    EXCEPTION
    WHEN OTHERS THEN
DBMS_OUTPUT.PUT_LINE('Some inner block error');
    END;

 EXCEPTION
 WHEN NO_DATA_FOUND THEN
DBMS_OUTPUT.PUT_LINE('No data found in fname');

 WHEN OTHERS THEN
DBMS_OUTPUT.PUT_LINE('Some outer block error');
 END;
```

Example 4: PL/SQL Block with Exception Handling

2.5 Debug the above block to trace the flow of control.
Additionally one can make appropriate changes in Select statement defined in the block to check the flow.

2.6: 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 • Implement business logic using Database Programming like Procedures and Functions • Implement validations in Procedures and Functions • Working with Packages • Performance Tuning
Time	4Hrs

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

3.1Write a PL/SQL block to find the maximum salary of the staff in the given department. Note: Department code should be passed as parameter to the cursor.

3.2. Write a function to compute age. The function should accept a date and return age in years.

3.3. Write a procedure that accept staff code and update staff name to Upper case. If the staff name is null raise a user defined exception.

3.4 Write a procedure to find the manager of a staff. Procedure should return the following – Staff_Code, Staff_Name, Dept_Code and Manager Name.

3.5. 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

3.6. Write a procedure that displays the following information of all staff

Staff_Name Department Name Designation Salary Status

Note: - Status will be (Greater, Lesser or Equal) respective to average salary of their own department. Display an error message Staff_Master table is empty if there is no matching record.

3.7. 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

3.8. Create a procedure that accepts the book code as parameter from the user. Display the details of the students/staff that have borrowed that book and has not returned the same. The following details should be displayed

Student/Staff Code Student/Staff Name Issue Date Designation Expected Ret_Date

3.9. Write a package which will contain a procedure and a function.

Function: This function will return years of experience for a staff. This function will take the hiredate of the staff as an input parameter. The output will be rounded to the nearest year (1.4 year will be considered as 1 year and 1.5 year will be considered as 2 year).

Procedure: Capture the value returned by the above function to calculate the additional allowance for the staff based on the experience.
Additional Allowance = Year of experience x 3000
Calculate the additional allowance and store Staff_Code, Date of Joining, and Experience in years and additional allowance in Staff_Allowance table.

3.10. 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.

3.11

Tune the following Oracle Procedure enabling to gain better performance.

Objective: The Procedure should update the salary of an employee and at the same time retrieve the employee's name and new salary into PL/SQL variables.

```
 CREATE OR REPLACE PROCEDURE update_salary (emp_id NUMBER) IS
v_nameVARCHAR2(15);
v_newsal NUMBER;
 BEGIN
  UPDATE emp_copy SET sal = sal * 1.1
  WHERE empno = emp_id;

  SELECT ename, sal INTO v_name, v_newsal
  FROM emp_copy
  WHERE empno = emp_id;

  DBMS_OUTPUT.PUT_LINE('Emp Name:' || v_name);
  DBMS_OUTPUT.PUT_LINE('Ename:' || v_newsal);
 END;
```

Example 5: Oracle Procedure

3.12

Write a procedure which prints the following report using procedure:

The procedure should take deptno as user input and appropriately print the emp details.

Also display :

Number of Employees,TotalSalary,MaximumSalary,Average Salary

Note: The block should achieve the same without using Aggregate Functions.

Sample output for deptno 10 is shown below:

```
Employee Name :  CLARK
```

```
Employee Job : MANAGER
Employee Salary :  2450
Employee Comission :
************************************
Employee Name :  KING
Employee Job : PRESIDENT
Employee Salary :  5000
Employee Comission :
************************************
Employee Name :  MILLER
Employee Job : CLERK
Employee Salary :  1300
Employee Comission :
************************************
Number of Employees :  3
Total Salary :  8750
Maximum Salary :  5000
Average Salary :  2916.67
-------------------------------------
```

Figure 1 :Report

3.13: Write a query to view the list of all procedures ,functions and packages from the Data Dictionary.

Lab 4Case Study 1

Goals	Implementation of Procedures/Functions ,Packages with Testing and Review
Time	2.5hrs

Consider the following tables for the case study.

Customer_Masters

Name	Null?	Type
Cust_Id	Not Null	Number(6)
Cust_Name	Not Null	Varchar2(20)
Address		Varchar2(50)
Date_of_acc_creation		Date
Customer_Type		Char(3)

Note: Customer type can be either IND or NRI

Account_Masters Table

Name	Null?	Type
Account_Number	Not Null	Number(6)
Cust_ID		Number(6)
Account_Type		Char(3)
Ledger_Balance		Number(10)

Note: Account type can be either Savings (SAV) or Salary (SAL) account.
For savings account minimum amount should be 5000.

Transaction_Masters

Name	Null?	Type
Transaction_Id	Not Null	Number(6)
Account_Number		Number(6)
Date_of_Transaction		Date
From_Account_Number	Not Null	Number(6)
To_Account_Number	Not Null	Number(6)
Amount	Not Null	Number(10)
Transaction_Type	Not Null	Char(2)

Note: Transaction type can be either Credit (CR) or Debit (DB).

Procedure and function should be written inside a package.
All validations should be taken care.

4.1 Create appropriate Test Cases for the case study followed up by Self/Peer to Peer Review and close any defects for the same.

4.2Write a procedure to accept customer name, address, and customer type and account type. Insert the details into the respective tables.

4.3. Write a procedure to accept customer id, amount and the account number to which the customer requires to transfer money. Following validations need to be done

- Customer id should be valid
- From account number should belong to that customer
- To account number cannot be null but can be an account which need not exist in account masters (some other account)
- Adequate balance needs to be available for debit

4.4 Ensure all the Test cases defined are executed. Have appropriate Self/Peer to Peer Code Review and close any defects for the same.

Lab 5 :Case Study 2

Goals	Implementation of Procedures/Functions ,Packages with Testing and Review
Time	2.5hrs

Consider the following table (myEmp) structure for the case study

EmpNo Ename City Designation Salary

The following procedure accepts Task number and based on the same performs an appropriate task.

```
PROCEDURE run_task (task_number_in IN INTEGER)
IS
BEGIN
  IF task_number_in = 1
  THEN
add_emp;
    --should add new emps in myEmp.
    --EmpNo should be inserted through Sequence.
    --All other data to be taken as parameters.Default location is Mumbai.
  END IF;
  IF task_number_in = 2
  THEN
raise_sal;
    --should modify salary of an existing emp.
    --should take new salary and empno as input parameters
    --Should handle exception in case empno not found
    --upper limit of rasing salary is 30%. should raise exception appropriately
  END IF;
   IF task_number_in = 3
   THEN
remove_emp;
          --should remove an existing emp
```

```
        --should take empno as parameter
    --Handle exception if empno not available
  END IF;
END run_task;
```

Example 66: Sample Oracle Procedure

However ,it has been observed the method adopted in above procedure is inefficient.

5.1

Create appropriate Test Cases for the case study followed up by Self/Peer to Peer Review and close any defects for the same.

5.2

Recreate the procedure (run_task) which is more efficient in performing the same.

5.3

Also, create relevant procedures (add_emp ,raise_sal ,remove_emp)

with relevant logic (read comments)to verify the same.

5.4 Extend the above implementation using Packages

5.5)Ensure all the Test cases defined are executed. Have appropriate Self/Peer to Peer Code Review and close any defects for the same.

Lab 6 :Handling Files, DBMS_LOB

Goals	Working with UTL_FILE and subprograms of this package
Time	1hr 30 mins

6.1: The following PL/SQL block creates file "TestFile.txt" with appropriate contents. Enhance the block by reading the contents back from the file and displaying it at SQL prompt.

```
Declare
        TextHandlerUtl_File.File_type;
        WriteMessageVarchar2(400);
        ReadMessageVarchar2(400);
Begin
        TextHandler:=Utl_File.Fopen('d:\Sample','TestFile.txt','W');
        WriteMessage:='FOPEN is a Function, which returns the value of type
        File_Type \n UTL_file.PUT_LINE is a procedure in UTL_FILE, which write a line
to a file,Specific line terminator will be appended \n';

        Utl_file.Putf(Texthandler,writemessage);
        Utl_File.Fflush(Texthandler);
        Utl_File.Fclose(Texthandler);

End;
/
```

Example 77: Block using File Handling operations

6.2 Extend the implementation in the above block by incorporating Exception Handling.

6.3 We need to maintain the above block in database permanently.What can be done for the same? Rewrite the above to achieve the same.

6.4: Write a PL/SQL block to
to read the contents from the file on the oracle server hard diskand display it on prompt. Ask the trainer for filename

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

Example 1: Declaration Block 5
Example 2: PL/SQL block 5
Example 3: PL/SQL block 7
Example 4: PL/SQL Block with Exception Handling 8
Example 5: Oracle Procedure 11
Example 6: Sample Oracle Procedure 16
Example 7: Block using File Handling operations 17