Usage Guidelines

Education and Research

Do not forward this document to any non-Infosys mail ID. Forwarding this document to a non-Infosys mail ID may lead to disciplinary action against you, including termination of employment.

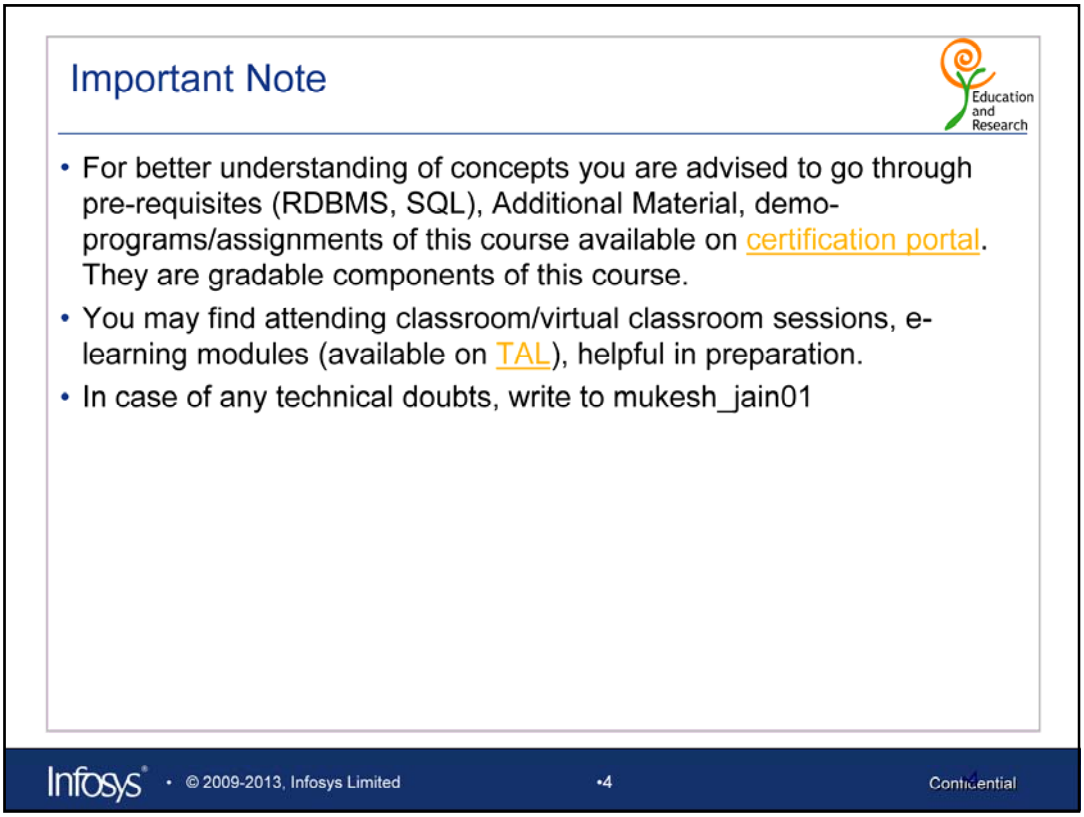
Contents of this material cannot be used in any other internal or external document without explicit permission from E&R@infosys.com.

•ES-TECH-P200-ESOR-ORA-PL-SQL

Oracle PL/SQL Programming

Confidential Information

- *This Document is confidential to Infosys Technologies Limited. This document contains information and data that Infosys considers confidential and proprietary ("Confidential Information").*
- *Confidential Information includes, but is not limited to, the following:*
  - *Corporate and Infrastructure information about Infosys;*
  - *Infosys' project management and quality processes;*
  - *Project experiences provided included as illustrative case studies.*
- *Any disclosure of Confidential Information to, or use of it by a third party, will be damaging to Infosys.*
- *Ownership of all Infosys Confidential Information, no matter in what media it resides, remains with Infosys.*
- *Confidential information in this document shall not be disclosed, duplicated or used – in whole or in part – for any purpose without specific written permission of an authorized representative of Infosys.*
- *This document also contains third party confidential and proprietary information. Such third party information has been included by Infosys after receiving due written permissions and authorizations from the party/ies. Such third party confidential and proprietary information shall not be disclosed, duplicated or used – in whole or in part – for any purpose without specific written permission of an authorized representative of Infosys.*



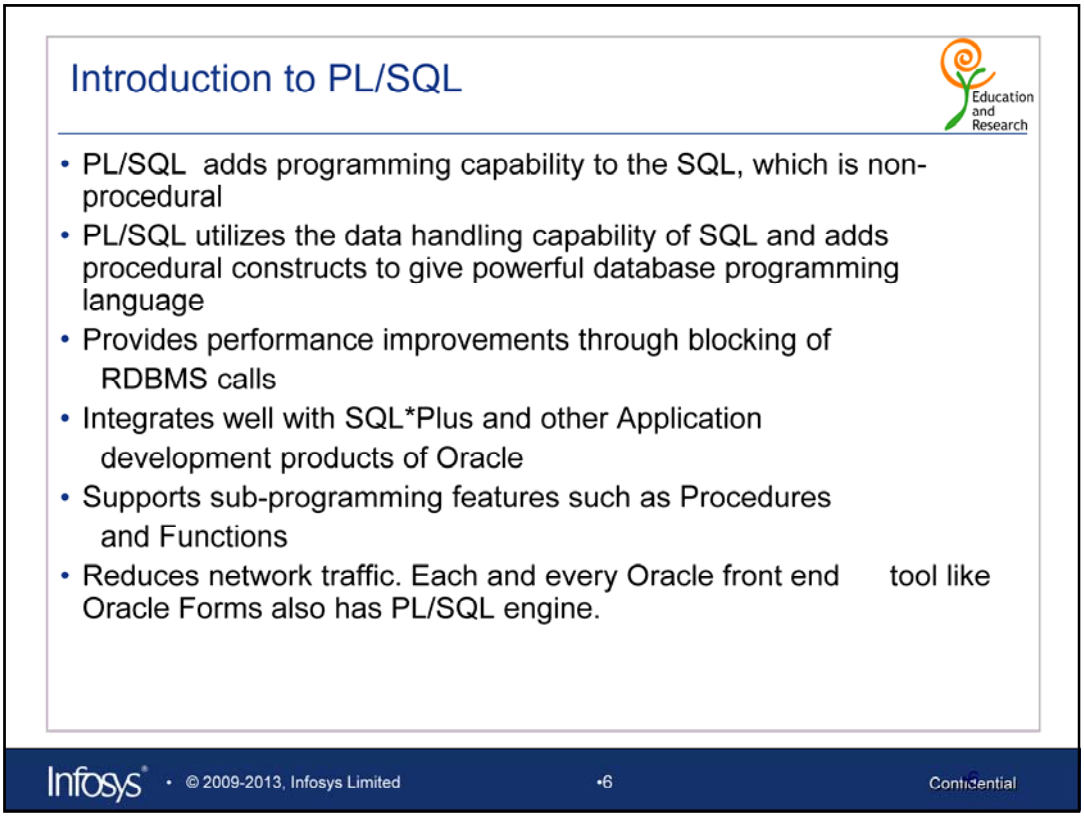
Certification material Download Link

Please refer the latest certification material for this course at

http://chddlf3014d:1111/sites/enr_portal_certifications/Lists/OS%20Technology%20101%20Download/AllItems.aspx

Session Plan

- Introduction to PL/SQL programming
- Usage of blocks, cursors and exceptions
- PL/SQL Program constructs
- Anonymous Blocks
- Variables and Data Types
- Anchored Declarations
- Conditional control statements
- Iterative control statements
- Collections and Records
- Bulk Binding
- Cursors
- Exceptions
- PL/SQL Best Practices

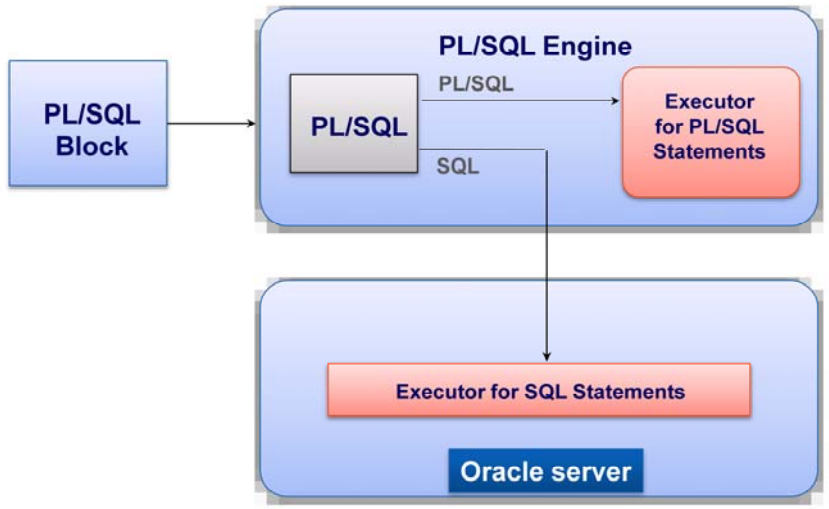


Introduction to PL/SQL

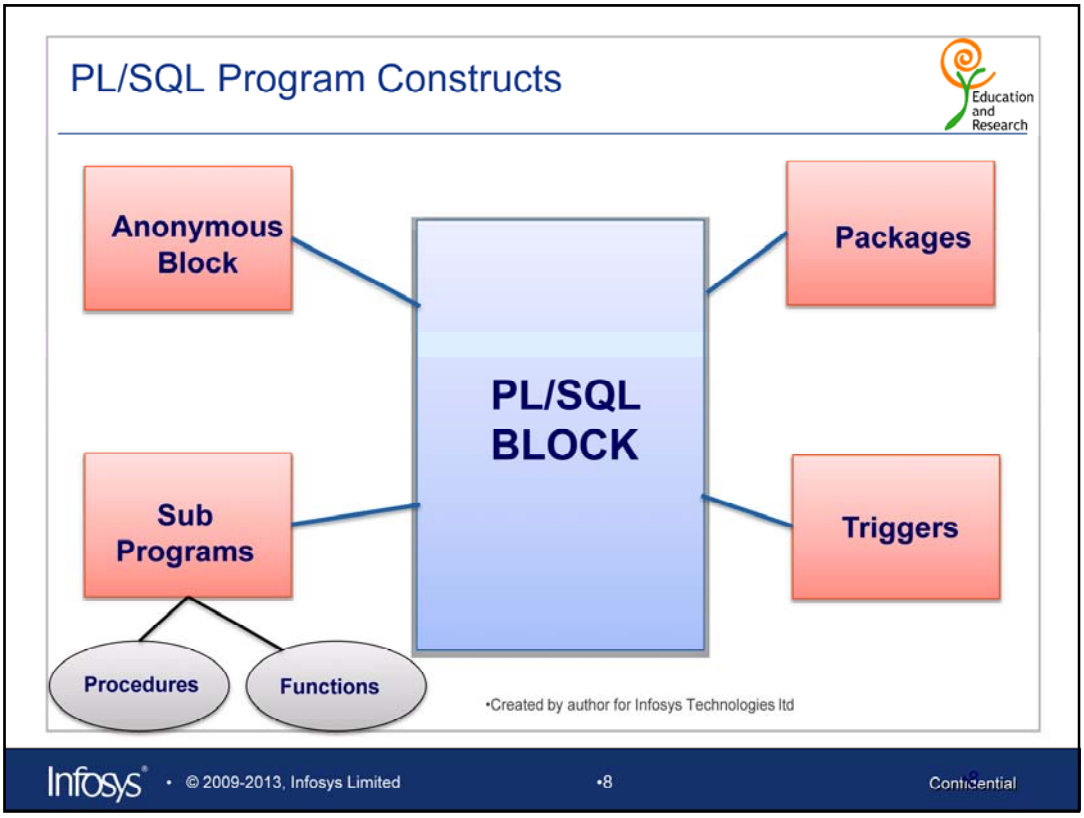
PL/SQL is a database programming language from Oracle, which extends SQL by adding procedural constructs to it. PL/SQL intends to fill the gap between database technology and procedural programming languages. Basically an application development tool, PL/SQL uses the sophisticated ORACLE RDBMS facilities, extends the standard SQL and is generally used to create *stored objects* and to add control functionality to the other ORACLE products like oracle forms developer.

Stored Objects refers to information that resides or is part of the database. ex. Tables, Views, etc.

PL/SQL Environment



•Created by author for Infosys Technologies ltd



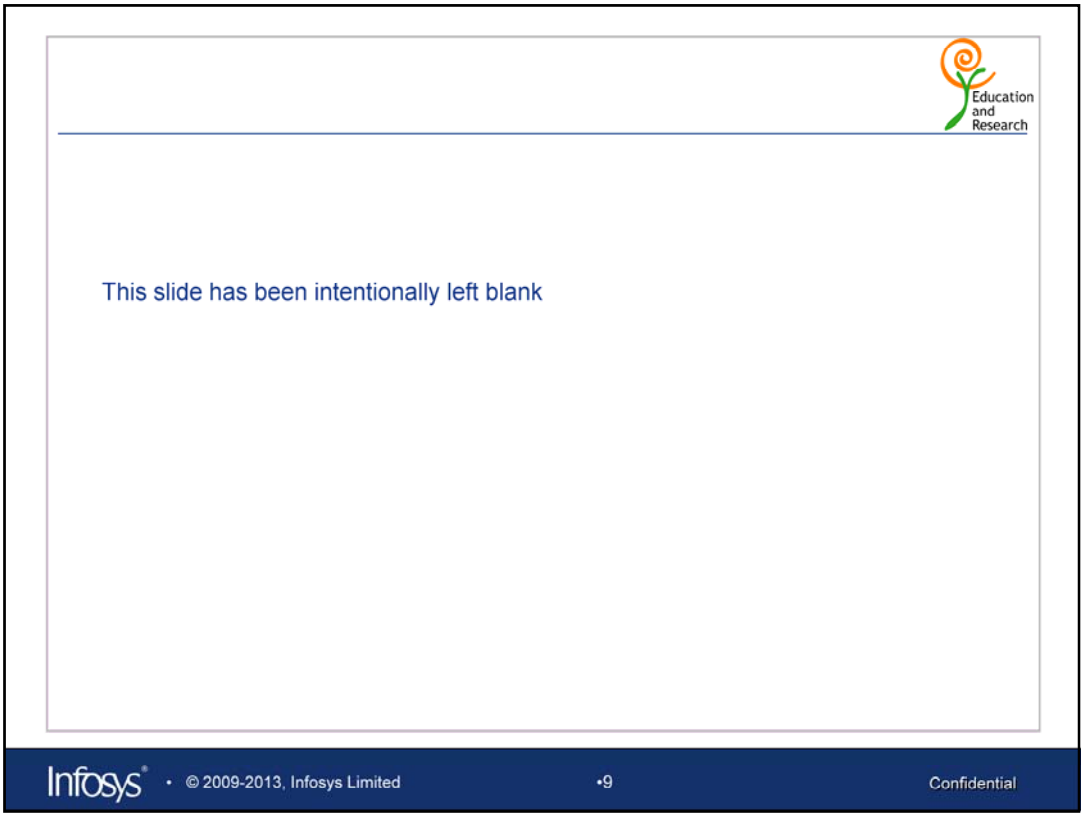
PL/SQL Program Constructs

Anonymous Block

- PL/SQL block without a name for identification
- It has to be brought to the server's memory every time it is required.It can't be accessed/referred neither by oracle nor by user for execution
- Has to be compiled every time we need that block of statements.Both compilation and execution are done at the server end.

Stored Procedures

- Named PL/SQL Block
- Stored as Database Objects like Tables, Views, Indexes, Sequences, Synonyms
- Stored in the ready to executable form
- Can be referred by other PL/SQL block
- One copy is brought to the buffer for sharing by multiple users



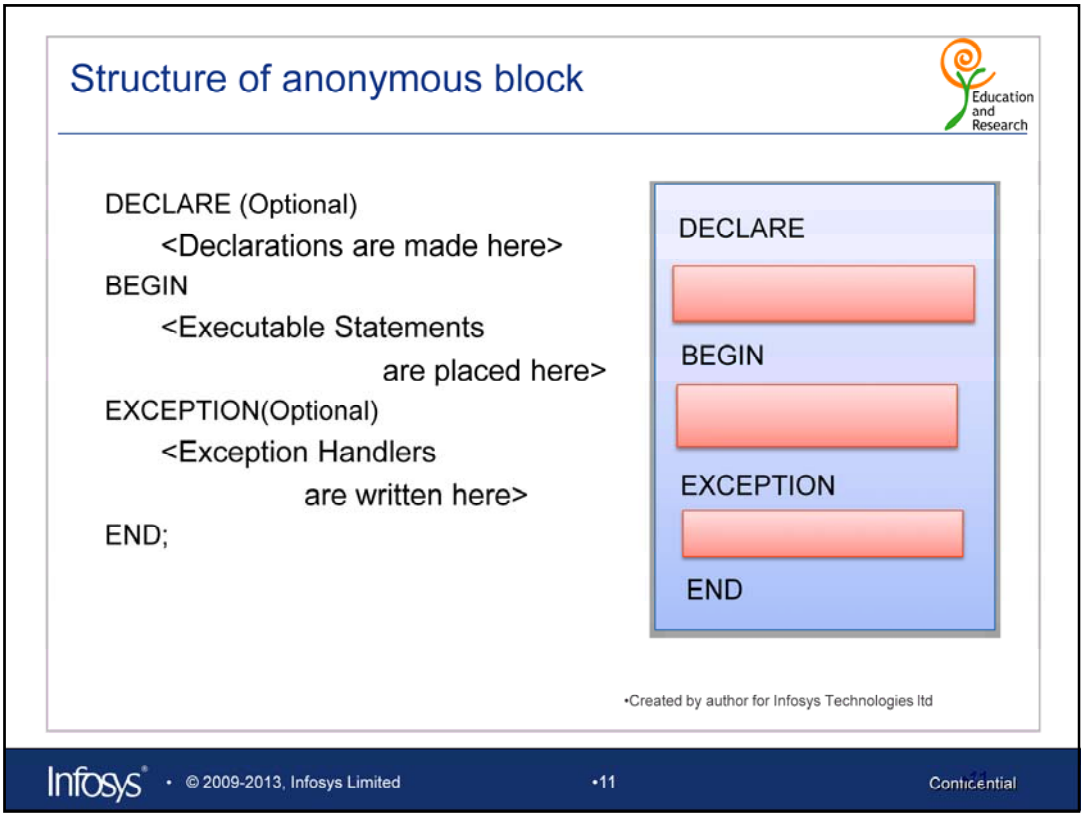
Packages

- ➢Collection of related objects
- ➢The components can be variables, Procedures or Functions
- ➢Collectively identified as one Database object

Triggers

- ➢The PL/SQL Block which gets fired automatically whenever some specified event happens
- ➢The chain of actions to be performed can be written in the trigger body for continual execution
- ➢Will be invoked by oracle automatically

Anonymous Blocks

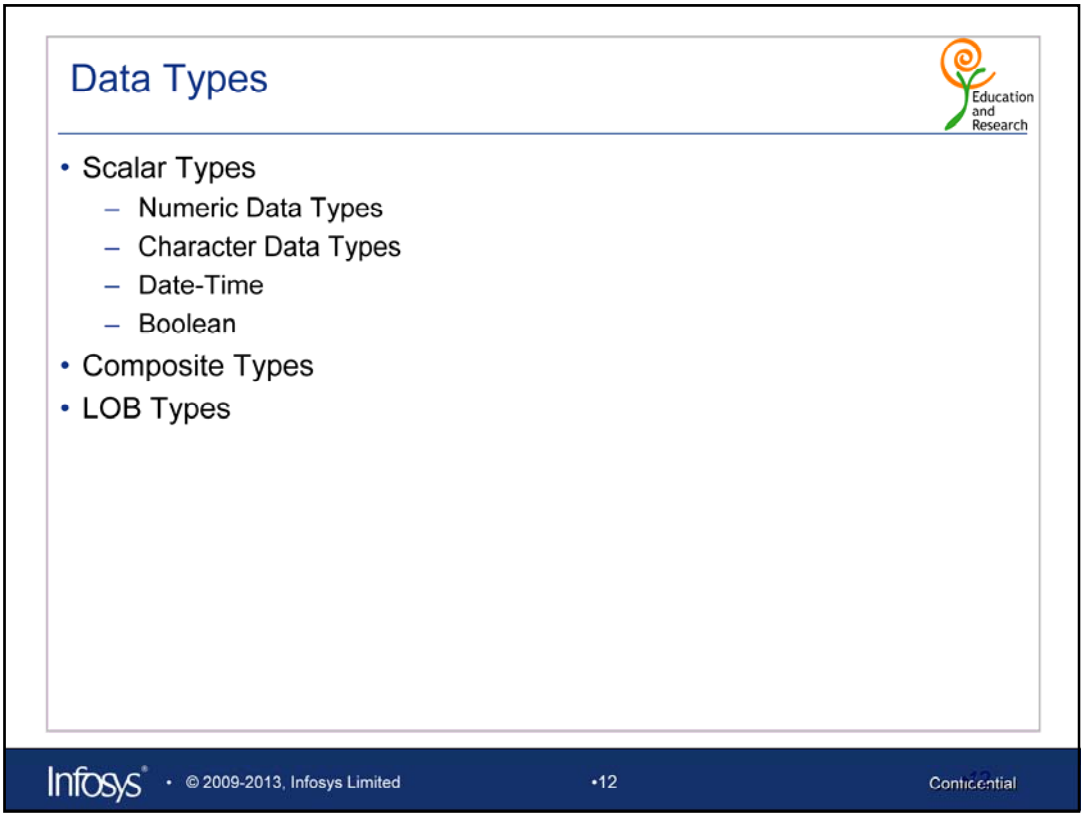


Structure of anonymous block

A standard PL/SQL code segment is called a *block*. Conceptually a block is made up of three sections:

1) A declaration section for variables, constants, exceptions and cursors, which is optional.
2) A section of executable statements, which is a must.
3) A section of exceptional statements or exception handlers, which is optional.

The order of the blocks is *logical.* The executable and exception handling sections can contain the nested blocks and not the other sections. Each block can contain nested blocks only in the executable or exception handling parts of the PL/SQL block. Once you nest a block the scoping rules for the variables, constants etc are physical from the point at which they are declared i.e. a variable defined in an inner block ceases to exist outside that block.



Data Types

PL/SQL offers a comprehensive set of predefined scalar data types. The variables are to be declared as part of the declare section using the following syntax,

<identifier name> <data type> :=<initialization>;

Scalar types are classified into four categories:

Number → BINARY_INTEGER, DECIMAL, FLOAT, INTEGER, NUMBER, REAL, etc.

Character → CHAR, VARCHAR2, LONG, etc.

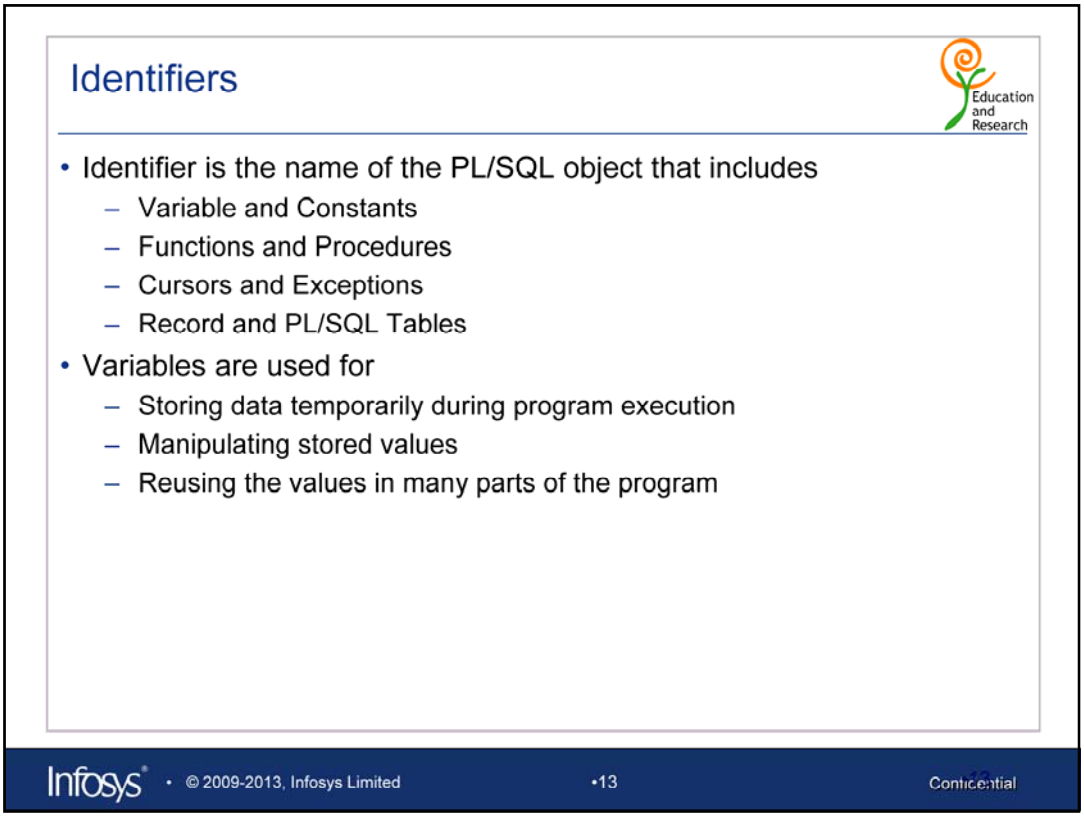
Boolean → BOOLEAN

Date-time → DATE

PL/SQL also supports following data types:

Composite → RECORD, TABLE, VARRAY

Large Object → CLOB, BLOB, NCLOB, BFILE



PL/SQL Identifier

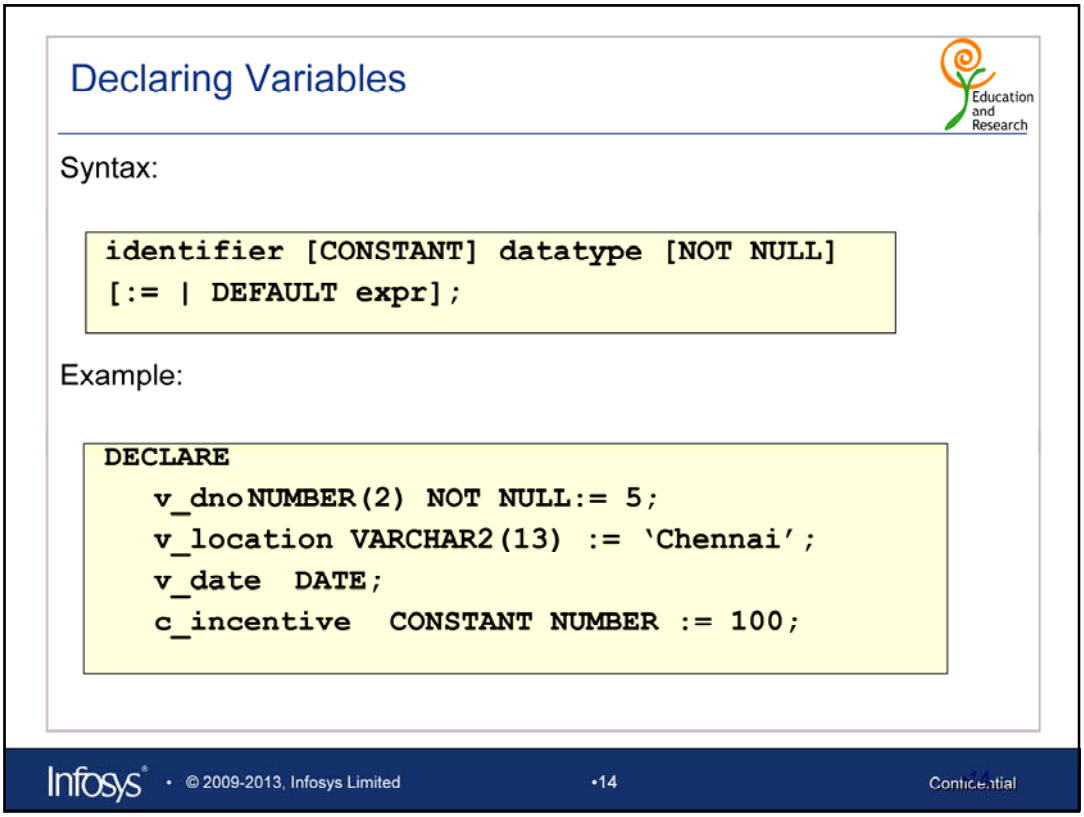
A PL/SQL identifier is a name of a PL/SQL object that includes:

Variable, Constant, Function, Procedure, Record, PL/SQL Table, Cursor, Exception, Package, etc.Properties of an identifierAn identifier

- Must start with an alphabet,Should not be of length more than 30 characterss
- Can include $, _ and #,Cannot contain spaces

Using Variables in PL/SQL: PL/SQL allows to declare variables and use them anywhere within the PL/SQL block, where an expression can be used. We can use variables in PL/SQL for the following purposes:

a) Storing data temporarily during program execution and use them for validating input or for later processing
b) Manipulating the stored values
c) Reuse: Reusing the same values in many parts of the program by referring them wherever required
d) Improved maintenance: Anchored variable declarations (variables declared using %TYPE and %ROWTYPE) can automatically change and adapt, if the underlying database columns change in their data type definitions.

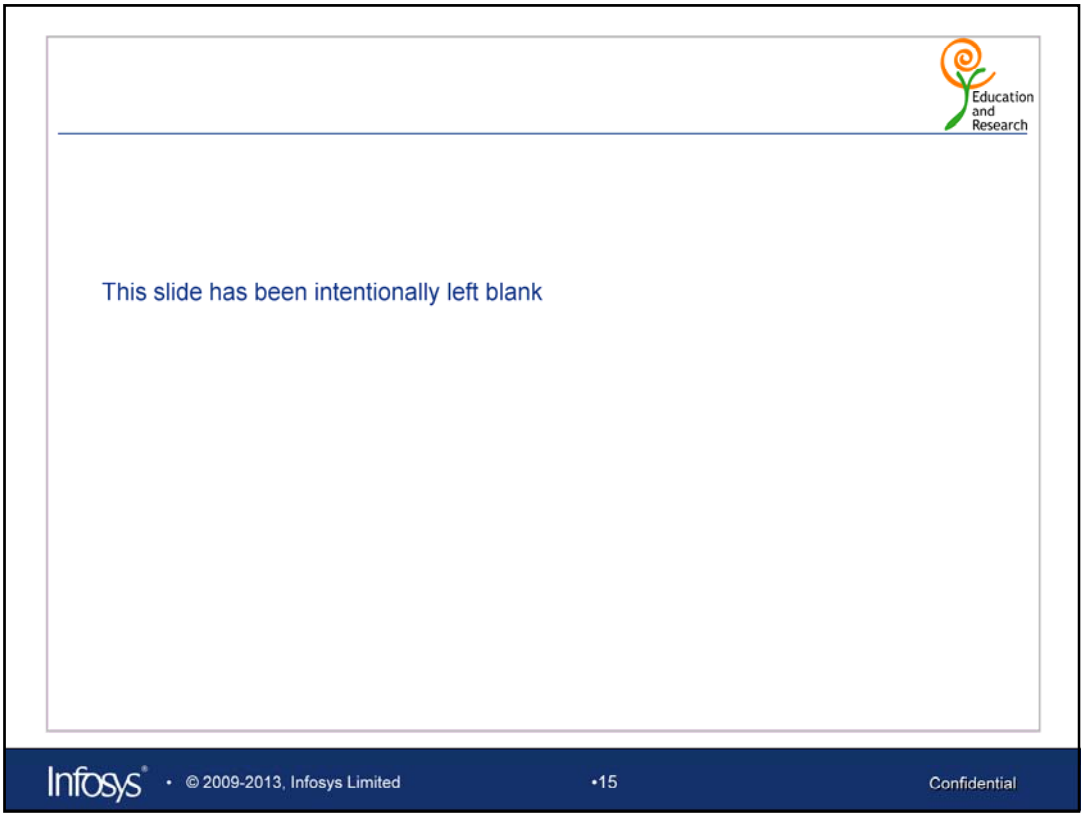


Using Variables in PL/SQL

Declaration

The PL/SQL variables can be declared and initialized in the declaration section of any PL/SQL block, subprogram or package. The variable declarations specify name and data type for the variable and allocate storage space according to their size. It is also possible to assign an initial value or default value to the variable using the DEFAULT keyword and can also impose a not null constraint on a variable using the NOT NULL keyword. The variables declared NOT NULL and the variables that are declared CONSTANT must compulsorily be followed by initialization.

PL/SQL does not allow forward references of variables and hence they need to be declared before they are referenced in any other statement, including other declarative statements.



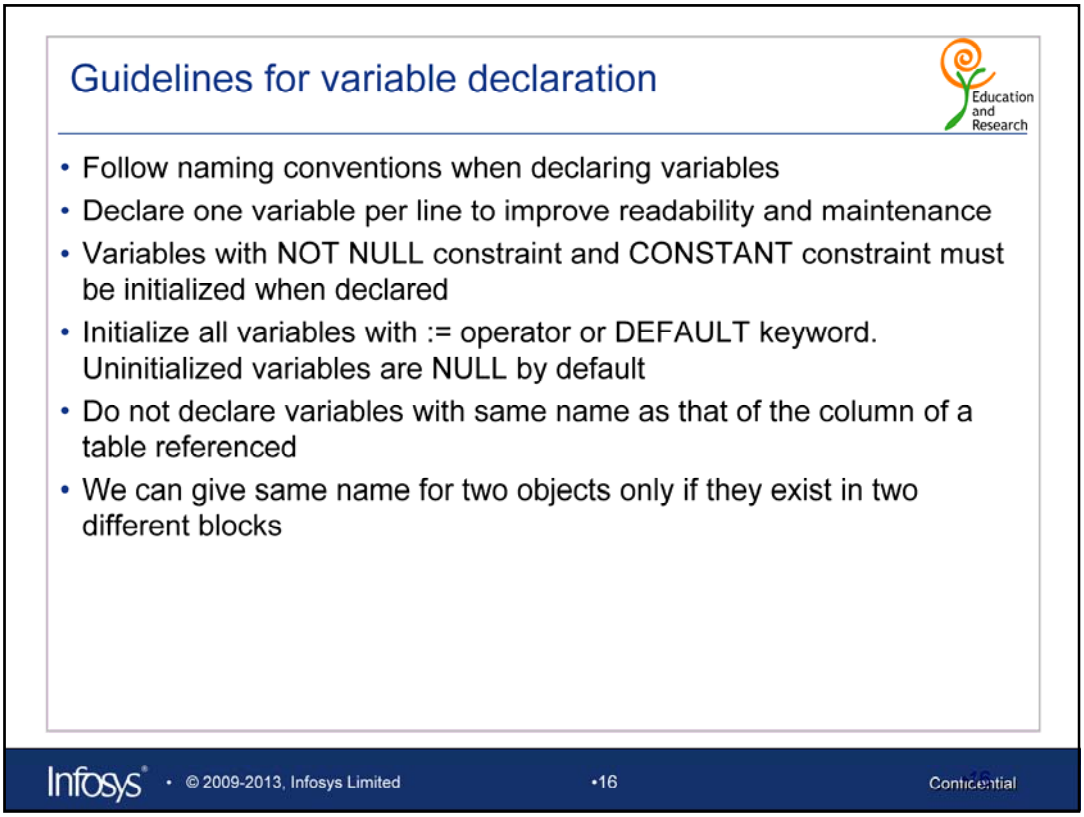
Assigning Values

The variables can be assigned values in the executable section of the PL/SQL block using := operator.

Passing Values into subprograms through parameters

Procedures and functions accept three types of parameters:

1) IN - Used to pass input to the subprogram like procedure or function
2) OUT - Used to read output from the subprogram like procedure or function
3) IN OUT - Used to pass an initial value and then get updated value in the same variable



Guidelines to be followed when declaring variables

➢Follow proper naming conventions when declaring variables. For example, v_name to represent a variable and c_name to represent a constant.

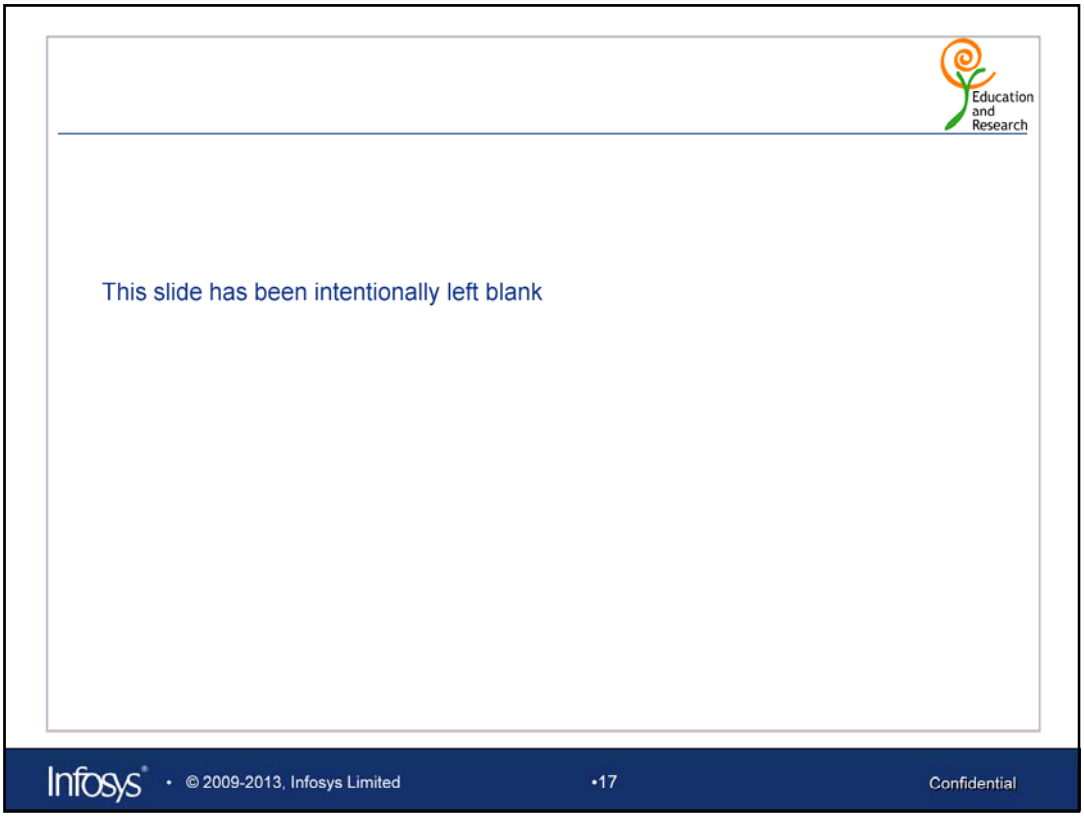
➢Declare only one identifier per line, which will make the code easier to read and maintain.

➢The variables with NOT NULL constraint and the constants must be assigned an initial value.

➢The CONSTANT keyword must precede the type specifier, when you declare the constants.

For example,

```
v_commission CONSTANT REAL := 500.00;
```



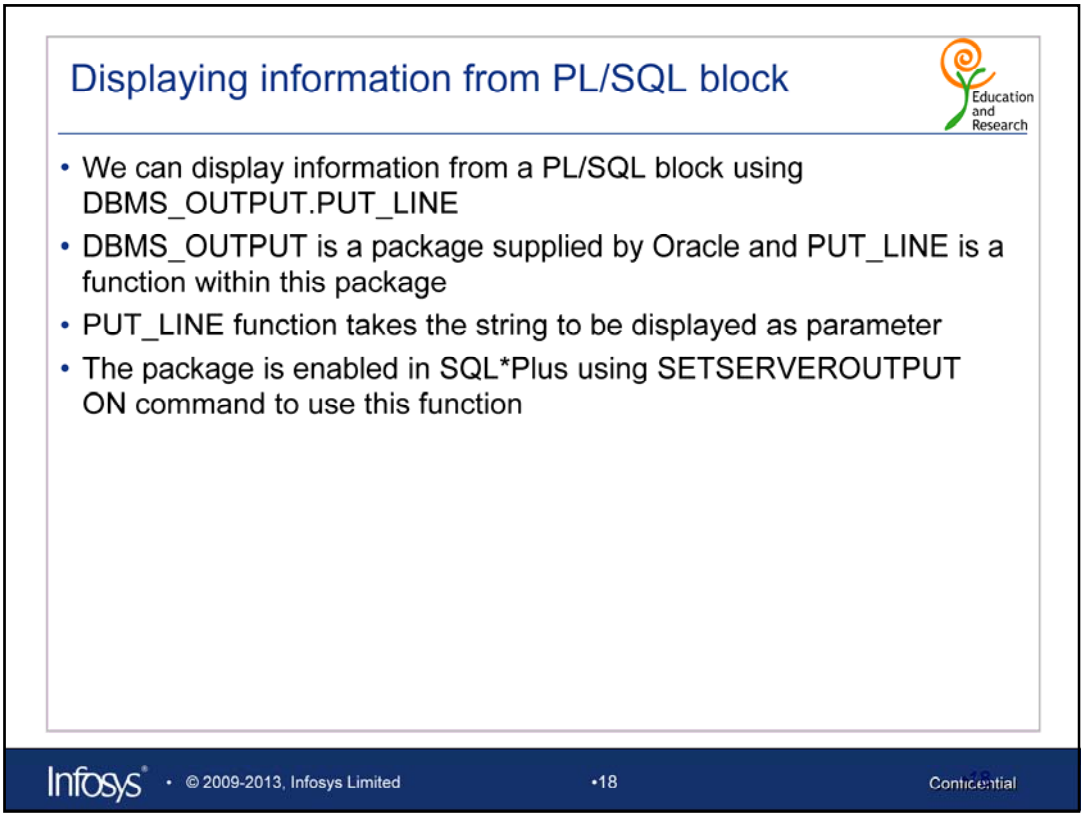
➢It is recommended to initialize all variables in the declare section using the assignment operator (:=) or DEFAULT keyword. Variables that are not initialized will contain NULL by default.

➢If a variable is declared with the same name as that of a column of a table used in the subprogram, then there will be ambiguity when the Oracle server accesses this variable. It may assume it to be the column of the table that is being referenced. Consider the following code for example,

```
DECLARE
  eno NUMBER(6);
BEGIN
  SELECT eno INTO eno FROM emp WHERE ename = 'Sanjay';
END;
```

This code may not work as expected in all the situations. Hence it is always better to follow proper naming conventions when naming the database objects.

➢We can have same name for the two object if they are in different blocks. Instead if one of them is present in the enclosed block, then the object which is declared in the current scope only will be considered.

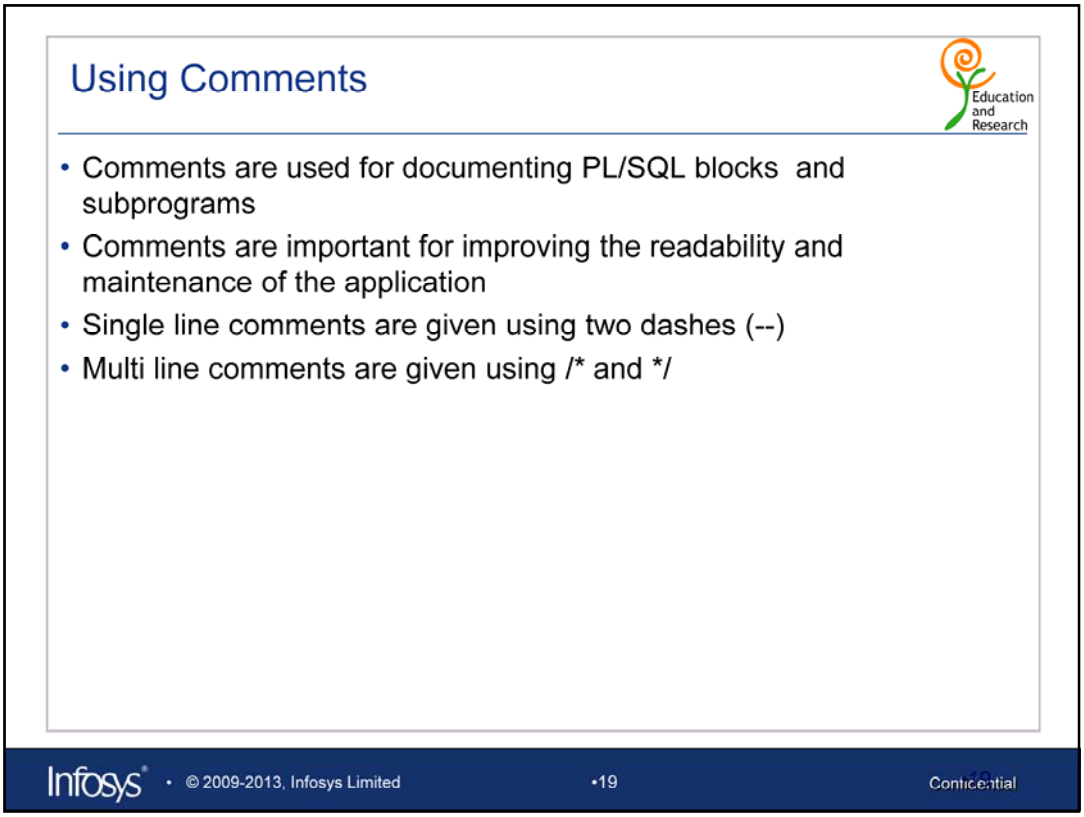


Displaying information from a PL/SQL block:

DBMS_OUPUT is a package supplied by Oracle. The PUT_LINE function of this package can be used for displaying information from a PL/SQL block. This function takes the string to be displayed as its parameter. But before using this package, it must be enabled in SQL*Plus using the command SET SERVEROUTPUT ON.

Example:

```
SET SERVEROUTPUT ON
DECLARE
  v_sal NUMBER(9,2) := 6000;
BEGIN
  v_sal := v_sal/12;
  DBMS_OUTPUT.PUT_LINE ('Salary of the employee
is||TO_CHAR(v_sal));
END;
```

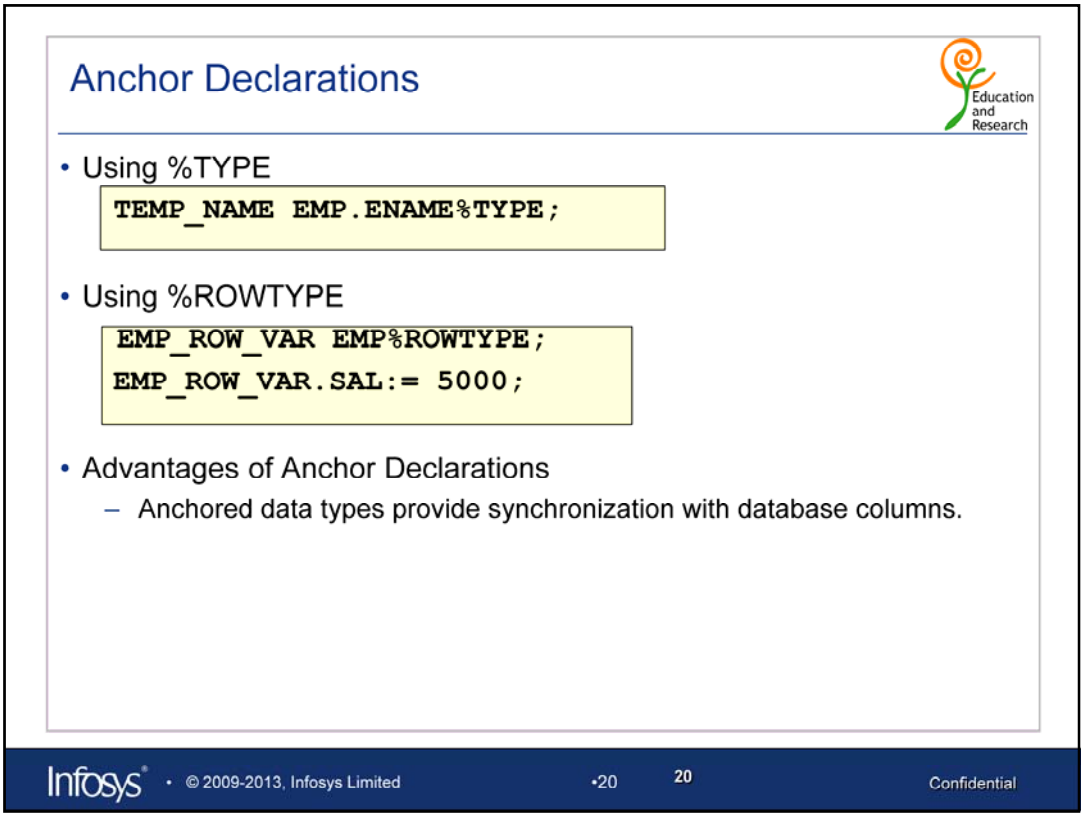


Commenting Code

Comments are used in PL/SQL blocks and subprograms to document the application and is very important for readability and maintenance of the application. There are two types of comments

a) Single line comments → Using two dashes (--) in the beginning of the line

b) Multi line comments → Using /* and */ by enclosing the lines between them



Anchored Declarations

PL/SQL provides two declaration attributes to anchor the data type of a variable to another PL/SQL variable of or a column of a table. Anchored data type can be declared using the following syntax,

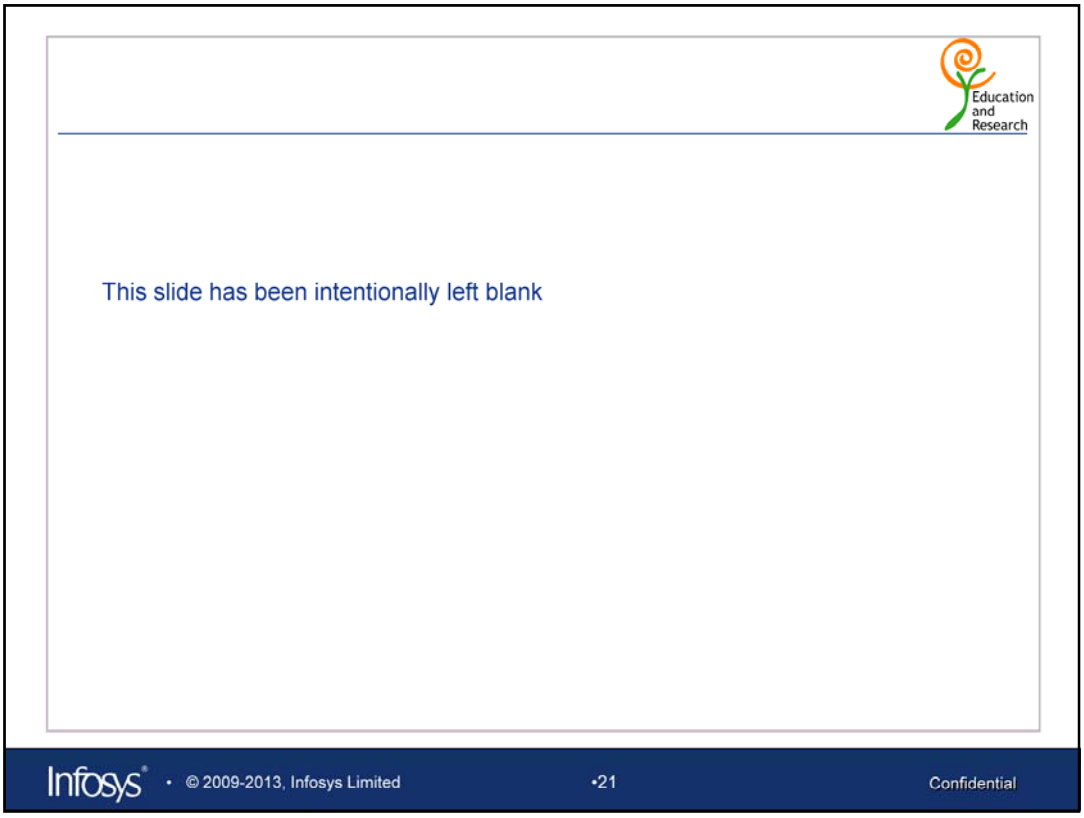
<variable_name> <type_attribute> %TYPE | %ROWTYPE [optional default value assignment]

- variablee_name → name of the variable that is being declared
- type_attribute → is any one of the following:

1) Previously declared PL/SQL variable name

Example:

tot_sales NUMBER(20,2);
mon_sales tot.sales%TYPE;



2) Table column in format 'table.column'

Example:

company_id company.com_id%TYPE;

3) If a PL/SQL variable is to refer to an entire record structure of the table then %ROWTYPE is used.

Example:

emp_rec emp%ROWTYPE;

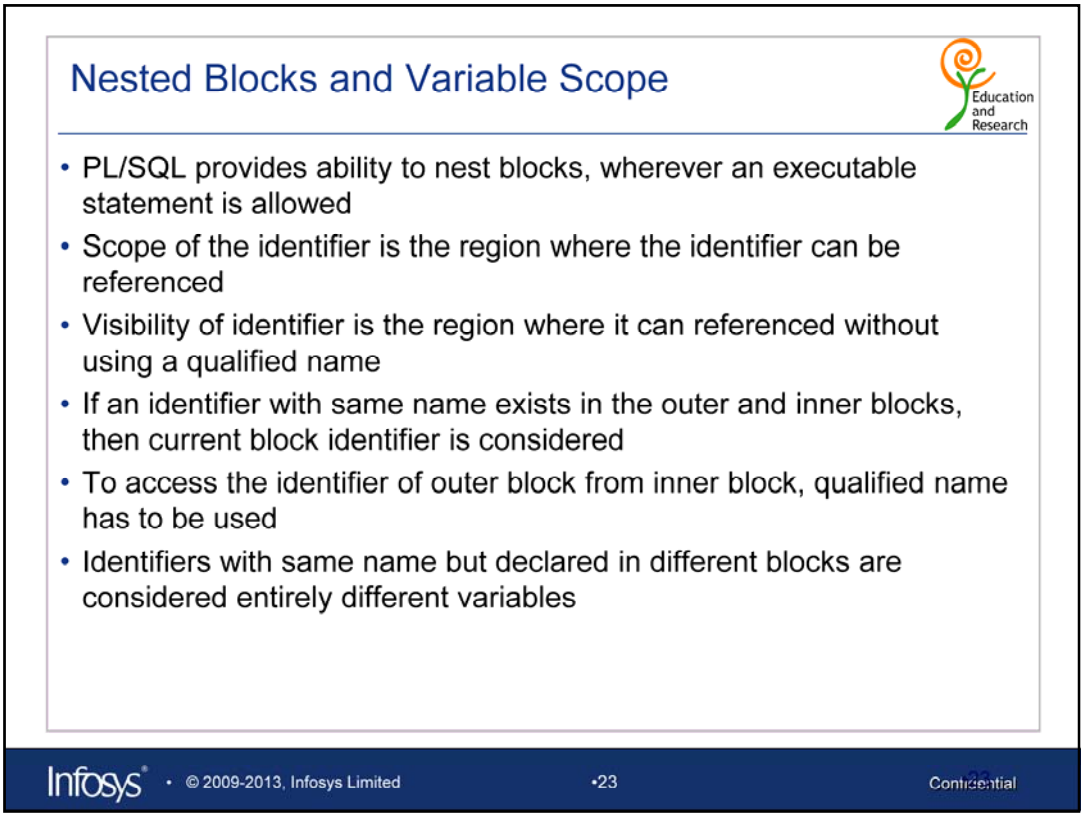
This declares a record of type equivalent to a row in the table emp. Accessing individual members of the record is by specifying emp_rec.*member*, where *member* refers to the column name in this case.

Tips when using variable declaration :

- Always code which establish a clear understanding of what data a variable is holding.
- Use named constants to avoid hard coding values.
- Remove unused variables from programs.
- Use %TYPE or %ROWTYPE when ever references are made to a database table.

Anchor Declarations - Example


```
DECLARE
lt_employeeNo  emp.empNo%TYPE;
lr_empRec emp%ROWTYPE;
BEGIN
lt_employeeNo := 60;
SELECT * INTO lr_empRec FROM emp WHERE EMP.EmpNo =
  lt_employeeNo;
  DBMS_OUTPUT.PUT_LINE ('Employee Name:
  '||lr_empRec.Name);
DBMS_OUTPUT.PUT_LINE ('Salary:||lr_empRec.sal);
END;
```

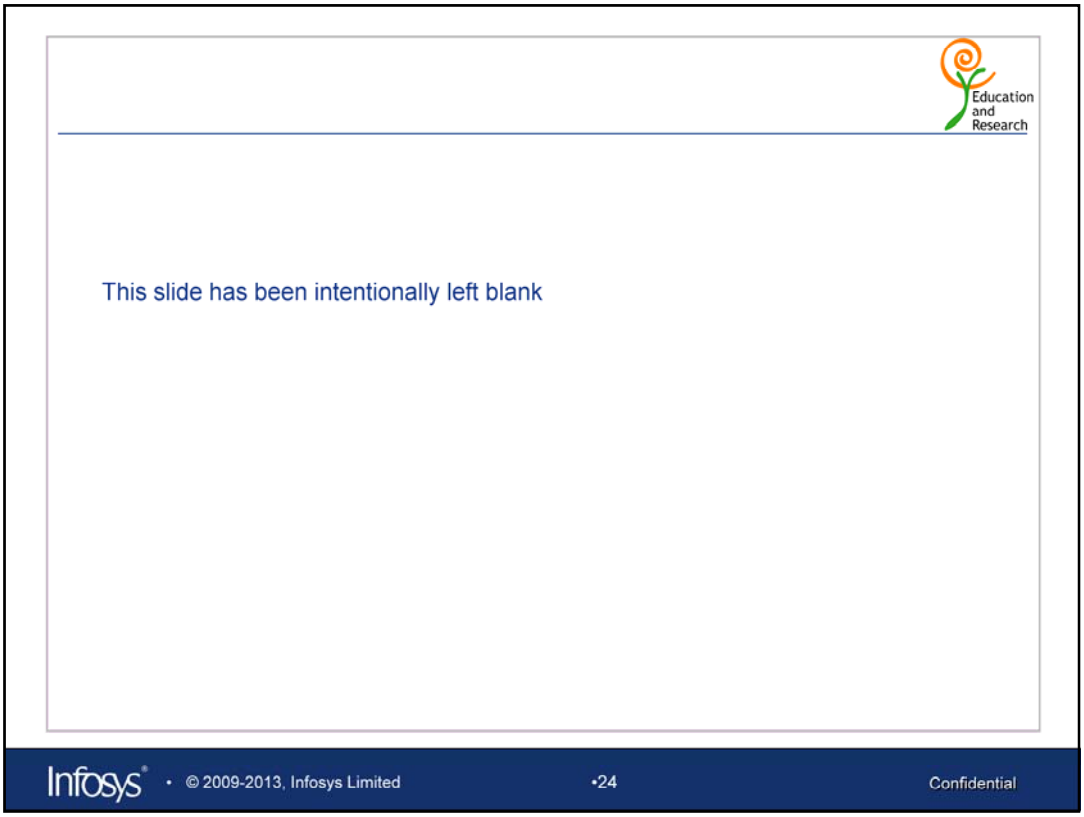


Nested Blocks

PL/SQL provides the ability to nest blocks and this can be done wherever an executable statement is allowed including the exception section. Therefore the executable section of the subprogram or PL/SQL block can be broken down to smaller blocks making it more readable.

Scope of the Identifiers

Scope of the identifier is the region of the block from which we can reference it. Any identifier declared in a block has a local scope to that block and has a global scope to all its sub-blocks. If a global identifier is re-declared in the sub block (that is an identifier with same name as that in the enclosing block is declared), the local identifier is considered when referenced. The global identifier can be referenced from the sub block only using a qualified name. But identifiers with same name but declared in different blocks are considered to be entirely different identifiers.



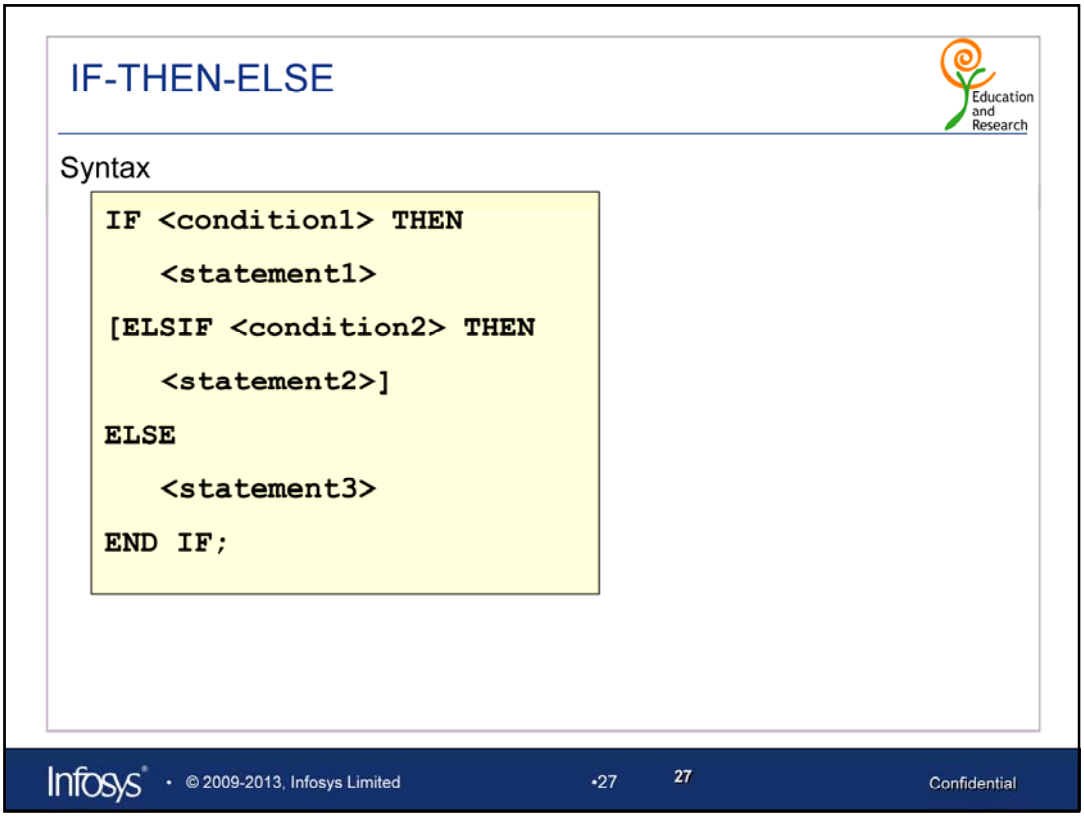
In nested blocks, a block can look up to the enclosing block for identifiers but a block cannot look down to the enclosed block for accessing identifiers. In the following example, y := x is valid whereas x := y is invalid.

```
x  BINARY_INTEGER;
BEGIN
DECLARE
   y  NUMBER;
 BEGIN
    y:= x;
 END;
END;
```

Conditions and Loops

Conditional and Iterative Control

- IF-THEN-ELSE Statement
- LOOP - END LOOP
- FOR – LOOP
- WHILE - LOOP



IF-THEN-ELSE: The IF statement is used for decision making based on a given condition. The syntax of the IF statement is,Type 1:

IF condition THEN

…..

END IF

The condition between the IF and THEN evaluates to true then code between the THEN and END IF (i.e. true part) will be executed. If condition evaluates to FALSE, then the code is not executed.

Type 2:

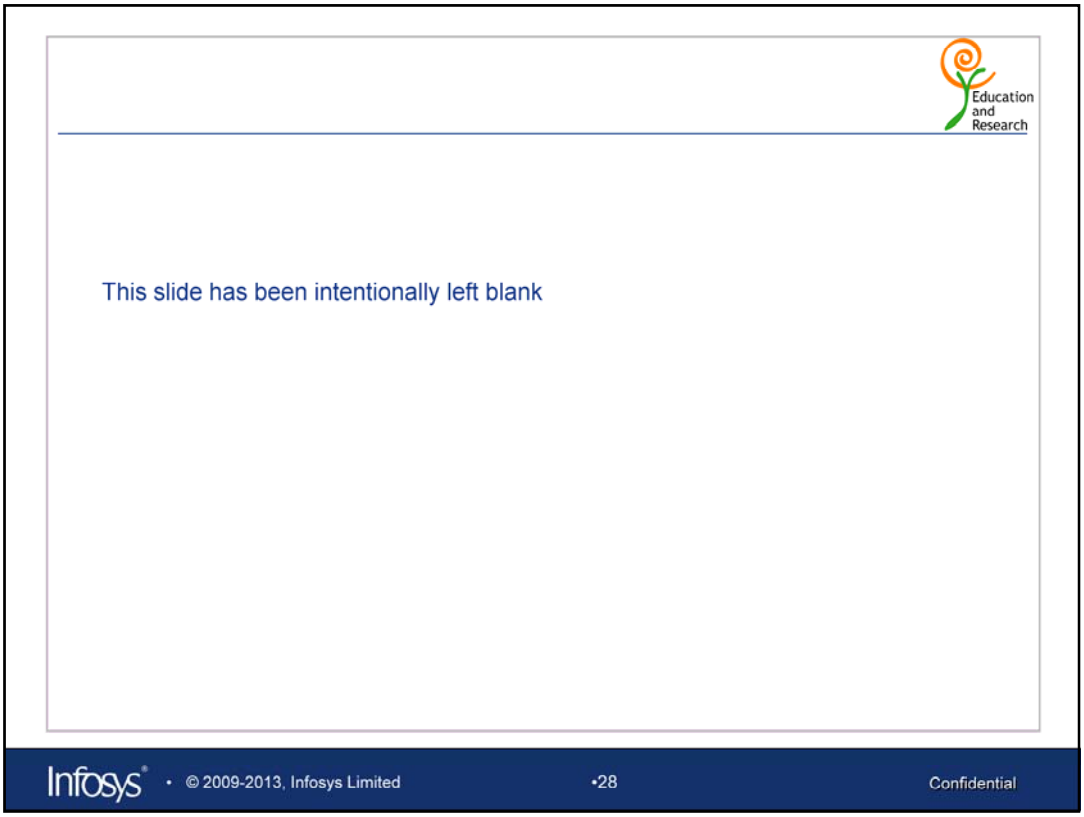
IF condition THEN

……..

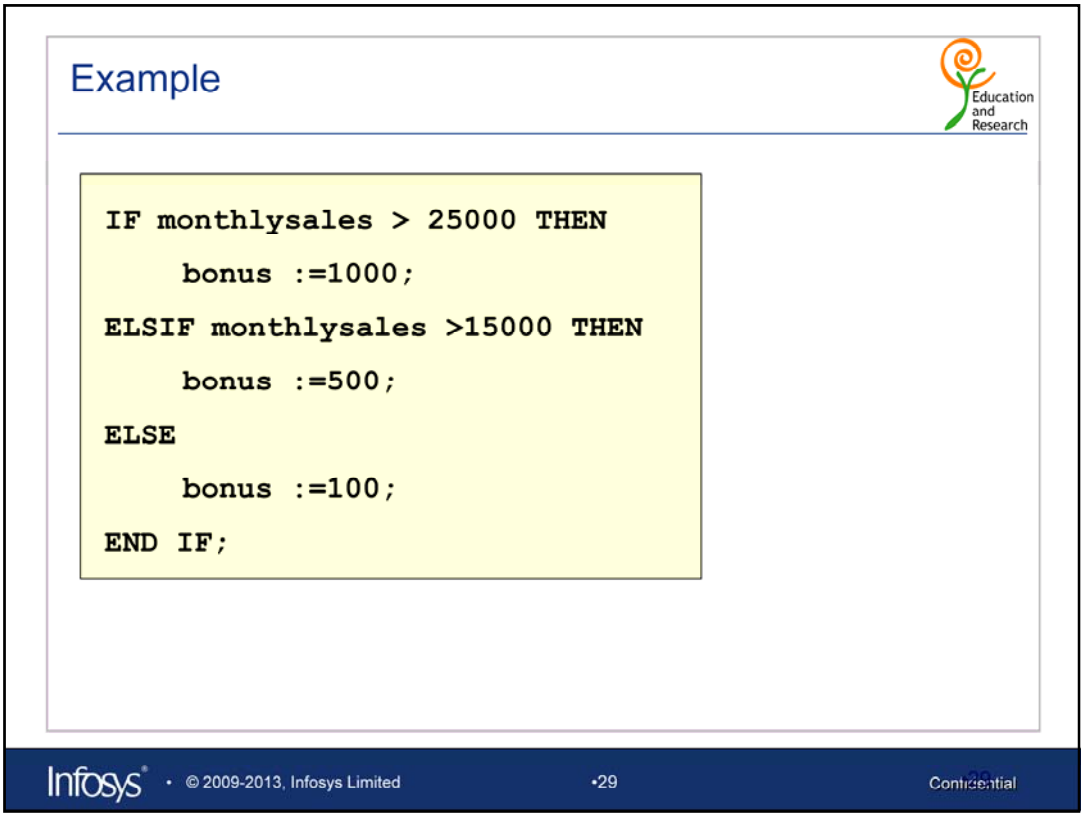
ELSE

…….

END IF;



The condition between the IF and THEN evaluates to true then code between the THEN and END IF (i.e. true part) will be executed. If condition evaluates to FALSE, then the code between ELSE and END IF (i.e. false part) will be executed.



IF-THEN-ELSE

Type 3:

IF condition THEN

……

ELSIF condition THEN

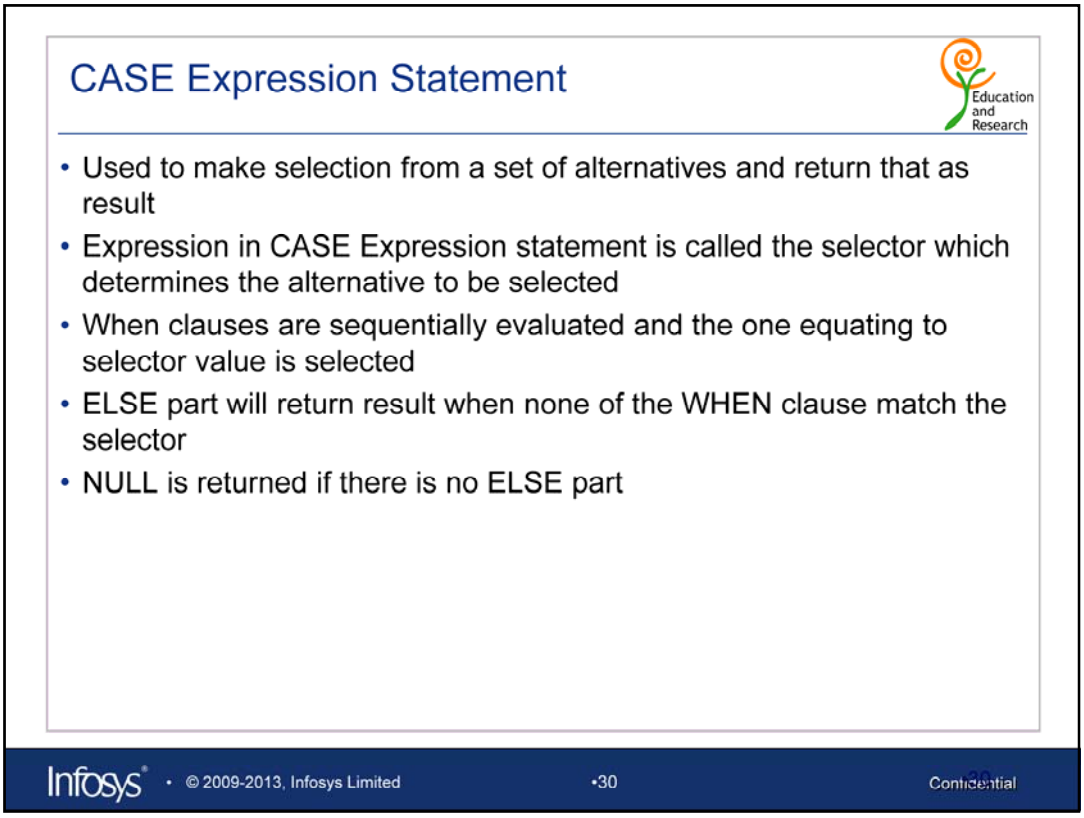
….

ELSE

….

END IF;

In this type of IF construct, a series of conditions are checked and either true or false part of the corresponding IF statement is executed based on the outcome of the condition.



Using CASE Expression statement

CASE Expression statement is used to make selection from a set of alternatives available and return that as result. The expression used in the CASE Expression statement is called the selector which is used to determine the alternative to be selected and returned.

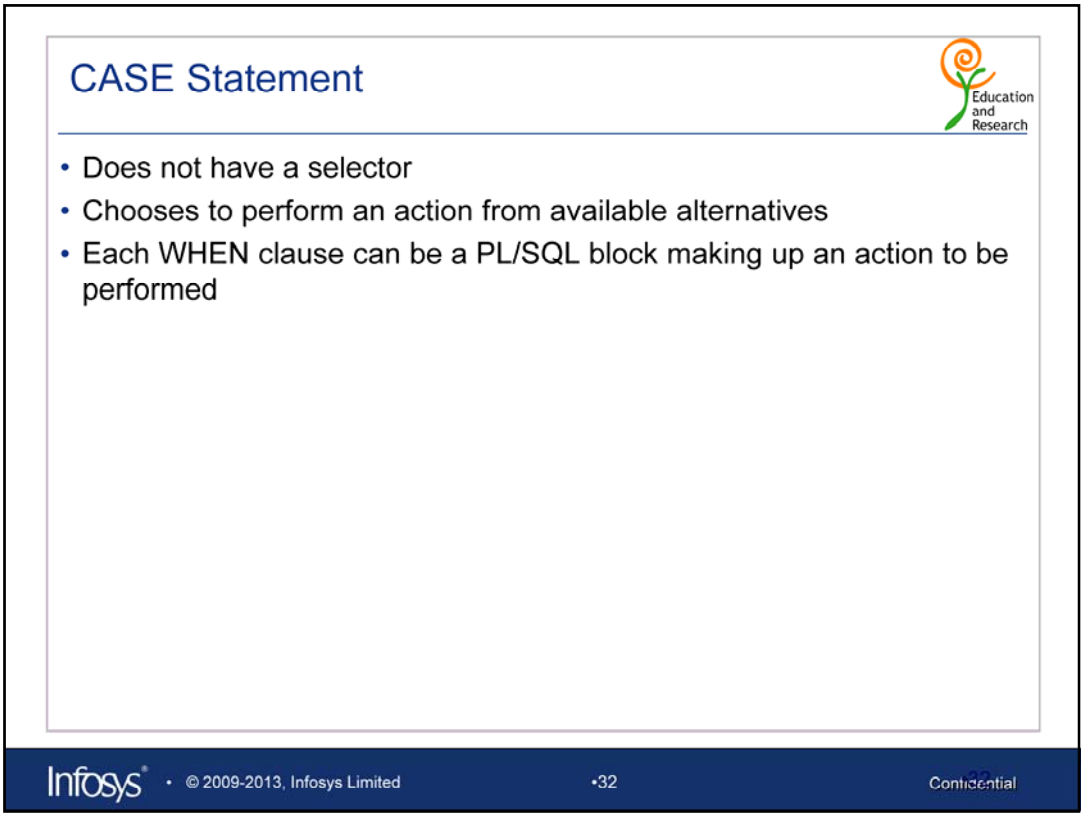
The syntax of the CASE Expression statement is:

```
CASE selector
  WHEN expression1 THEN result1
  WHEN expression2 THEN result2
  ...
  WHEN expression THEN result
 [ELSE resultN+1]
END;
```

The WHEN clauses are sequentially evaluated and the first WHEN clause whose expression evaluates to a value equal to that of the selector, is selected and the result corresponding to that WHEN clause is returned. The subsequent WHEN clauses are not evaluated. If none of the WHEN clauses are selected, then the ELSE part will return the result. If the ELSE part is not present, then NULL value is returned from the CASE Expression statement.

Example

```
SET SERVEROUTPUT ON

DECLARE
     v_empgrade      CHAR(1) := 'A';
     v_empappraisal VARCHAR2(25);

BEGIN
     v_empappraisal := CASE v_empgrade

  WHEN 'A' THEN 'Excellent Performance'
    WHEN 'B' THEN 'Very Good Performance'
    WHEN 'C' THEN 'Good Performance'
          ELSE 'No such grade'
  END;
  DBMS_OUTPUT.PUT_LINE ('Grade of Employee: ' ||
  v_empgrade || ' Appraisal' || v_appraisal);

END;
```



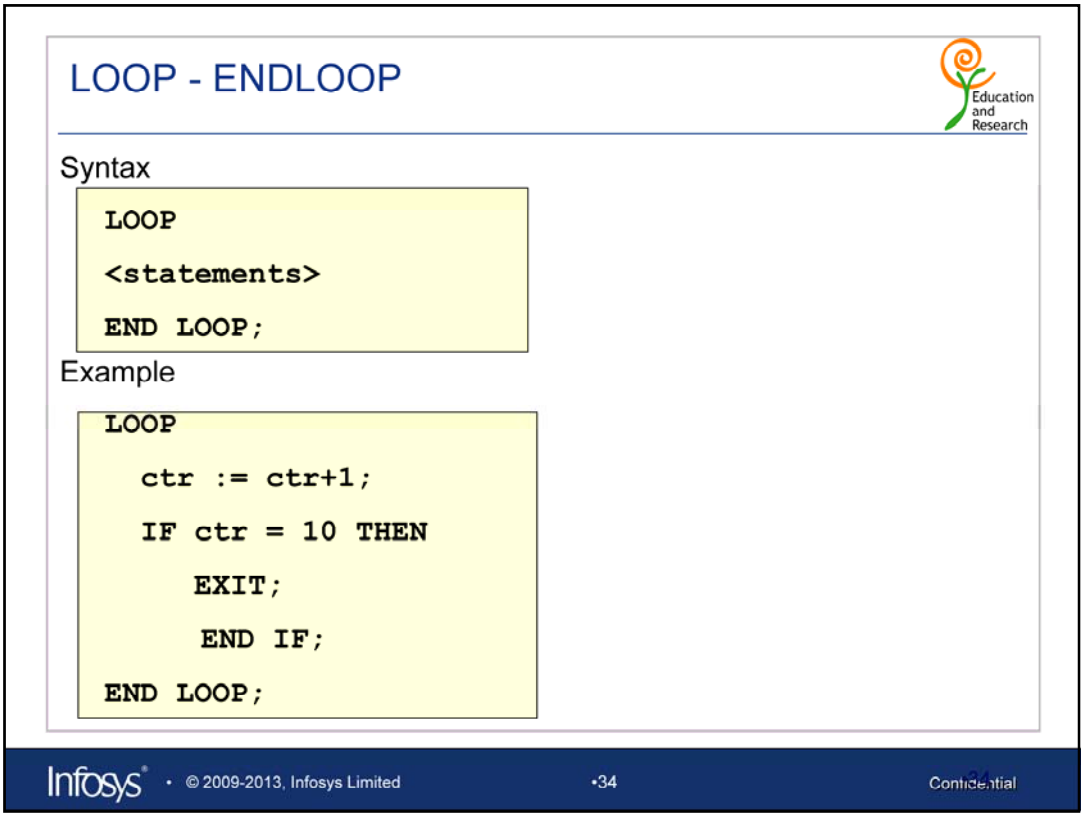
Using CASE Statement

Alternatively, the CASE statement can be used for scenarios where one of the alternative multiple available alternatives has to be selected. In the CASE statement, there is no selector and each WHEN clause can be an entire PL/SQL block.

```
CASE
  WHEN condition1 THEN result1
  WHEN condition2 THEN result2
  ...
  WHEN conditionN THEN result
 [ELSE resultN+1]
END;
```

Example

```
DECLARE
   empgrade CHAR(1); empappraisal VARCHAR2(20);
BEGIN
   appraisal := CASE
   WHEN empgrade = 'A'
          THEN dbms_output.put_line('Excellent
          Performance');
   WHEN empgrade = 'B'
          THEN dbms_output.put_line('Very Good
          Performance');
   WHEN empgrade = 'C'
          THEN dbms_output.put_line('Good
          Performance');
    ELSE dbms_output.put_line('No such grade');
END CASE;
```



Loops

Loops are used to execute a given set of statements repeatedly.

Three types of Loop constructs available in PL/SQL are:

The simple or infinite loop

The FOR loop

The WHILE loop

The simple or infinite LOOP

Syntax: LOOP

<executable statements>

END LOOP;

The body must consist of at least one executable statement.

Example: LOOP rem_balance := account_balance(accno);

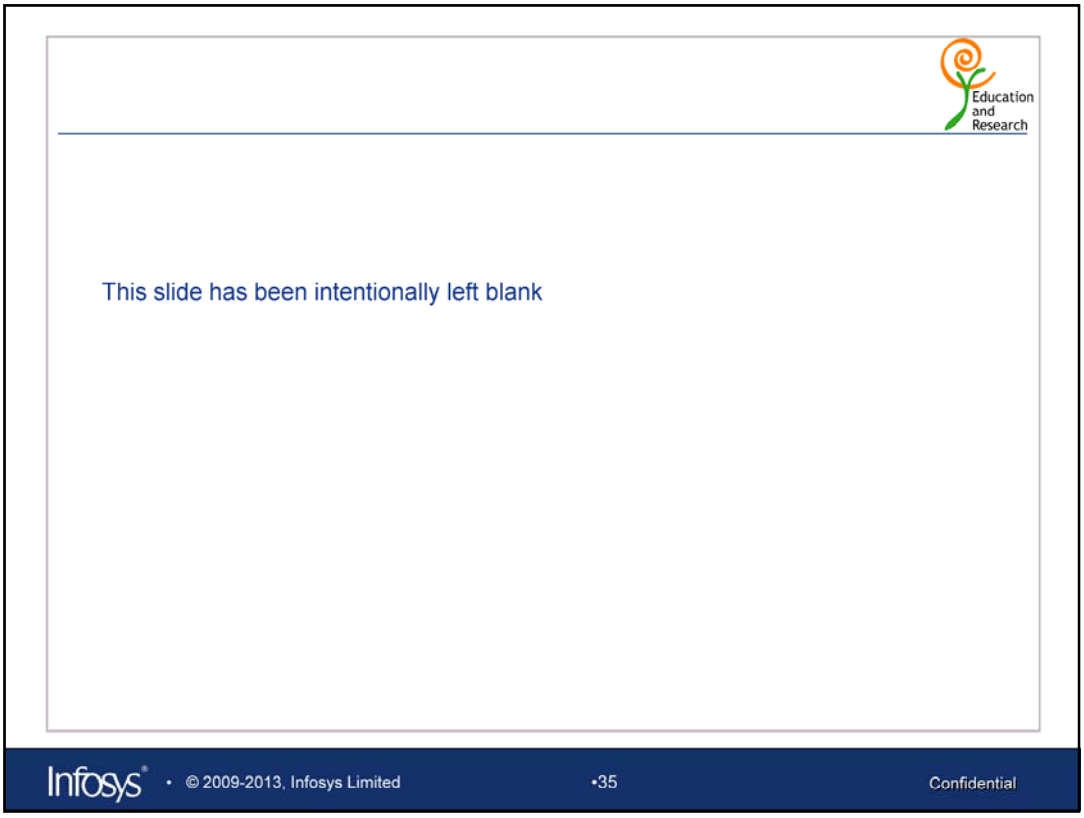
IF rem_balance < 100 THEN

EXIT;

ELSE apply_balance(accno,rem_balance);

END IF;

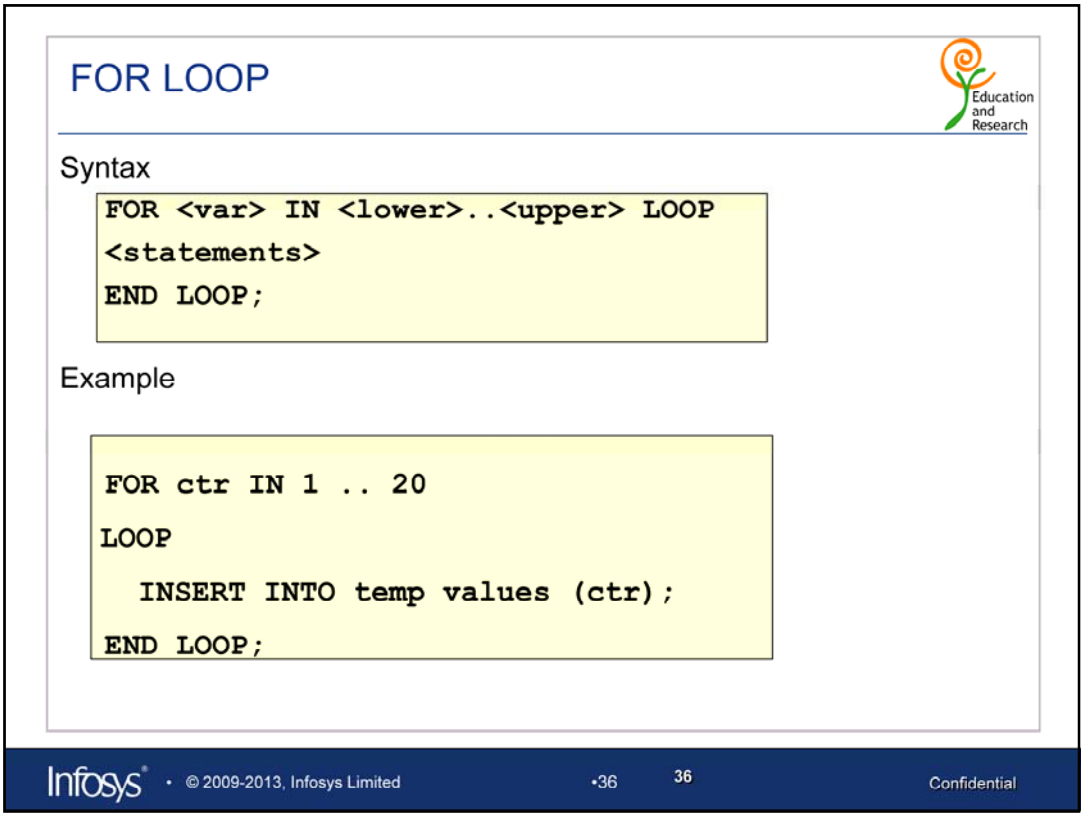
END LOOP;



EXIT statement inside a loop is used to terminate the loop. If the EXIT is to be executed after checking a condition, then we can use EXIT WHEN statement, which is an alternative to using IF-THEN with EXIT statement.

Example:

```
LOOP
        rem_balance := account_balance(accno);
         EXIT WHEN rem_balance < 100;
        apply_balance(accno,rem_balance);
END LOOP;
```



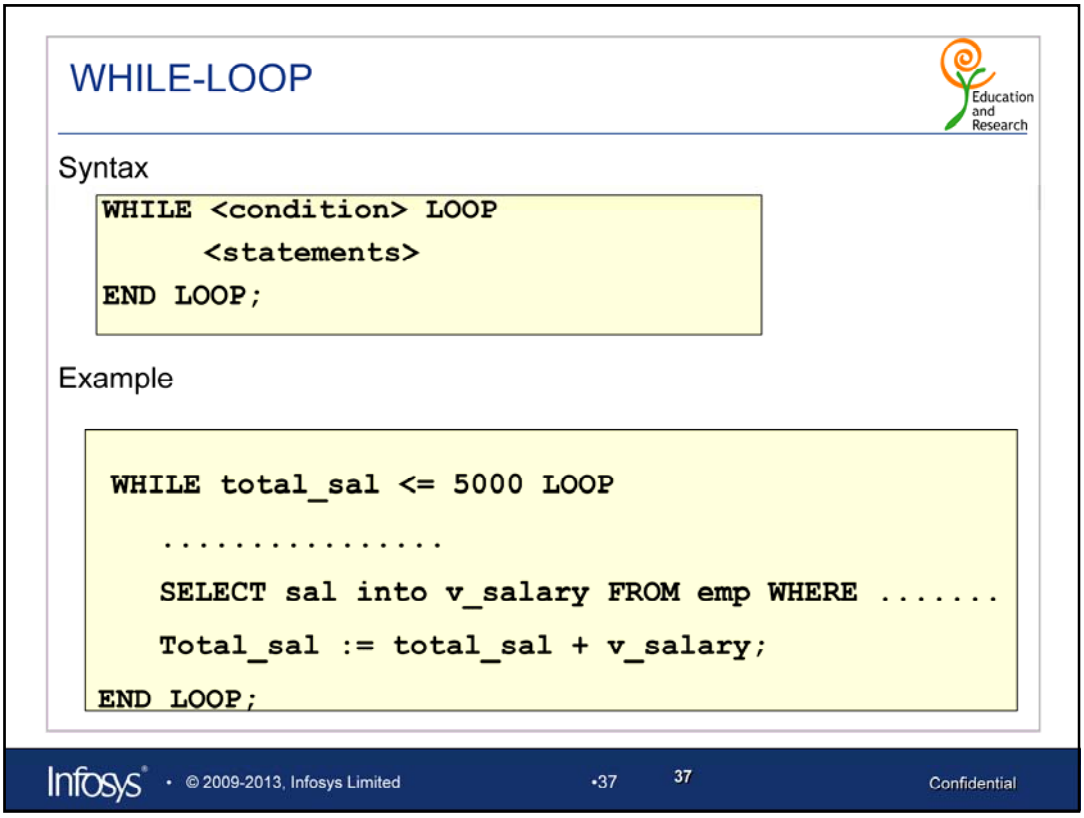
Using a FOR loop

The syntax of the FOR loop is:

FOR <loop_index> IN [REVERSE] <low_number>..<high_number>
LOOP
<executable statement(s)>
END LOOP;

Following points have to considered when using the FOR loop:

a) Loop index should not be declared. The loop index is scoped within the loop.
b) The evaluation of expressions in low_number and high_number happens only once at the start of the loop
c) The loop index or the low_number or the high_number should never be changed within the loop
d) Usage of EXIT statement in the FOR loop should be avoided
e) To perform iteration in reverse order of the range (i.e from high_number to low_number), use the REVERSE keyword. The range values should not be reversed.



WHILE Loop

The WHILE loop is used to iterate a set of statements repeatedly as long as the condition evaluates to true. Once the condition evaluates to false, the loop terminates.

Syntax:

WHILE <condition>
LOOP
<executable statements>
END LOOP;

Note:

- All the values required to evaluate the condition in WHILE loop should be known before the loop execution starts.
- If the condition is evaluated to false, before the loop starts, then the loop will not be executed even once.

Collections and Records

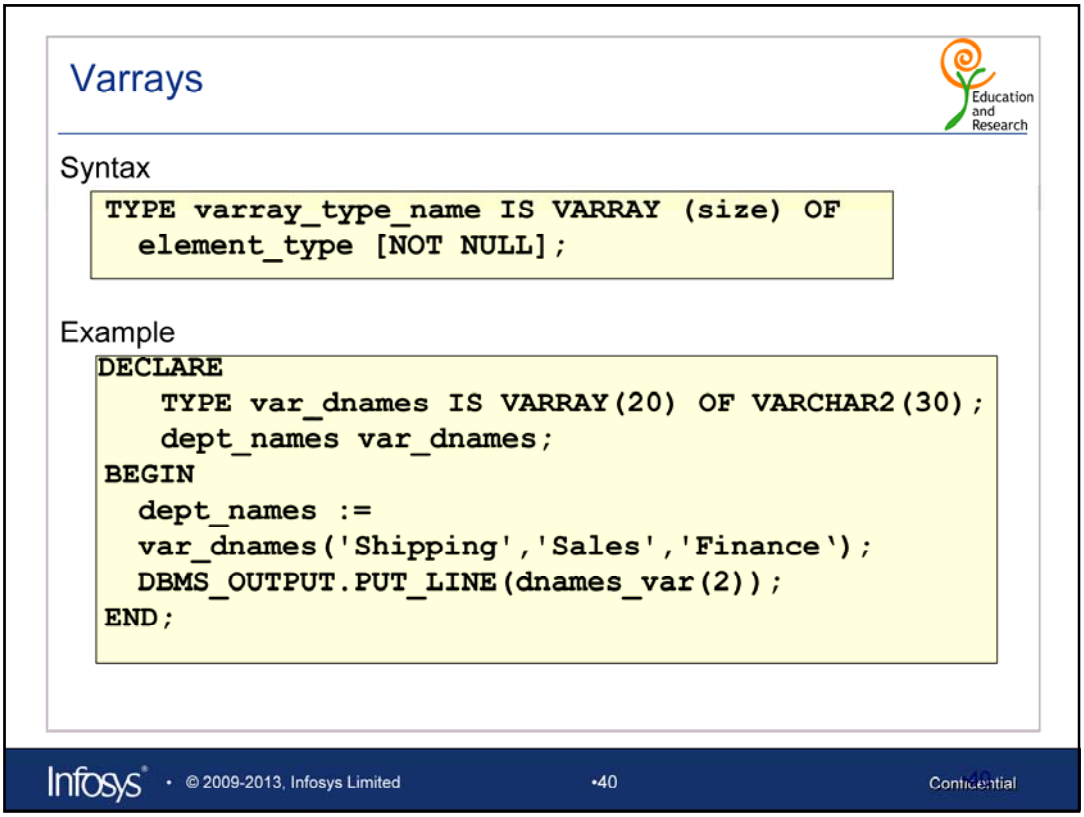
VARRAY, NESTED TABLE, RECORD

PL/SQL Collections

- Single dimensional data structures containing ordered group of elements with same data type
- Types of Collections
  1) Varray

     Collection of elements with lower bound and upper bound. Varrays are never sparse.
  2) Nested Table

     Collection of elements without any bounds. Nested tables can be sparse.



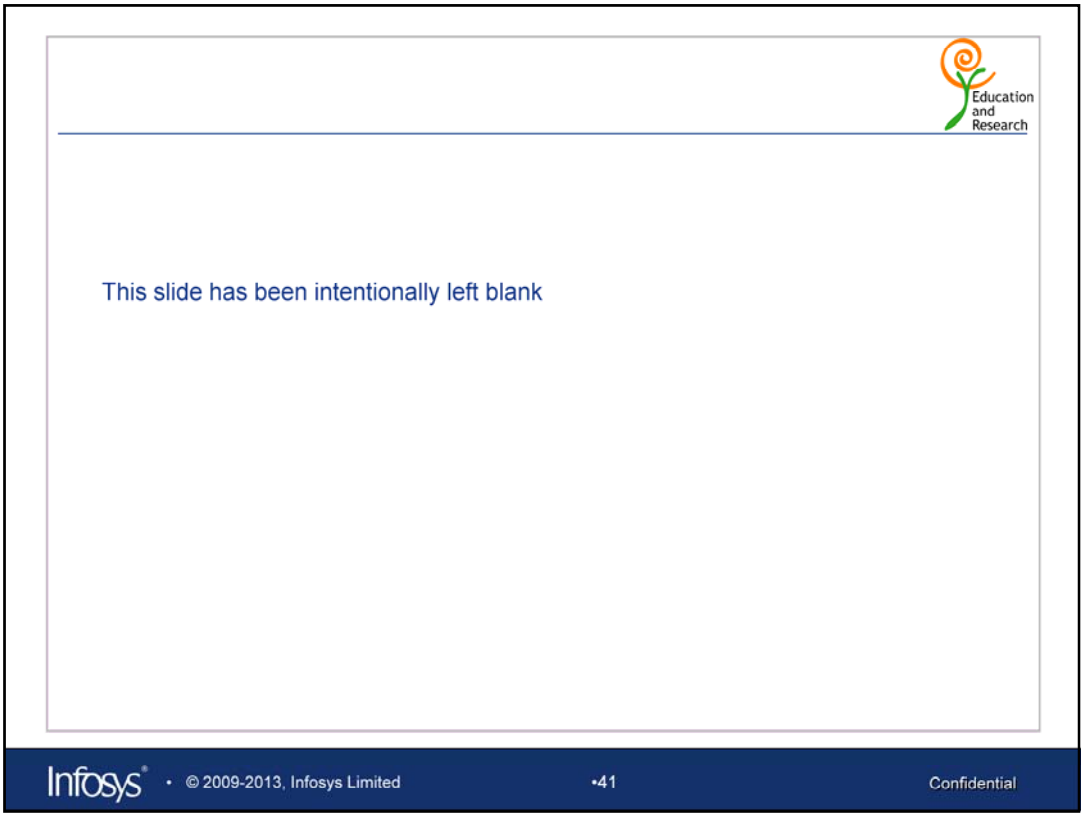
Varrays:Varrays are collection of elements of same data type. These have an upper bound within which their size can vary. These can never be sparse. That is there can never be empty element locations within the collection.

There are two steps in creating a varray collection.

Step 1:First the varray type has to be created specifying the size and data type of the elements to be stored in the varray. The syntax for creating a varray type is:TYPE varray_type_name IS VARRAY (size) OF element_type [NOT NULL];

In this syntax:

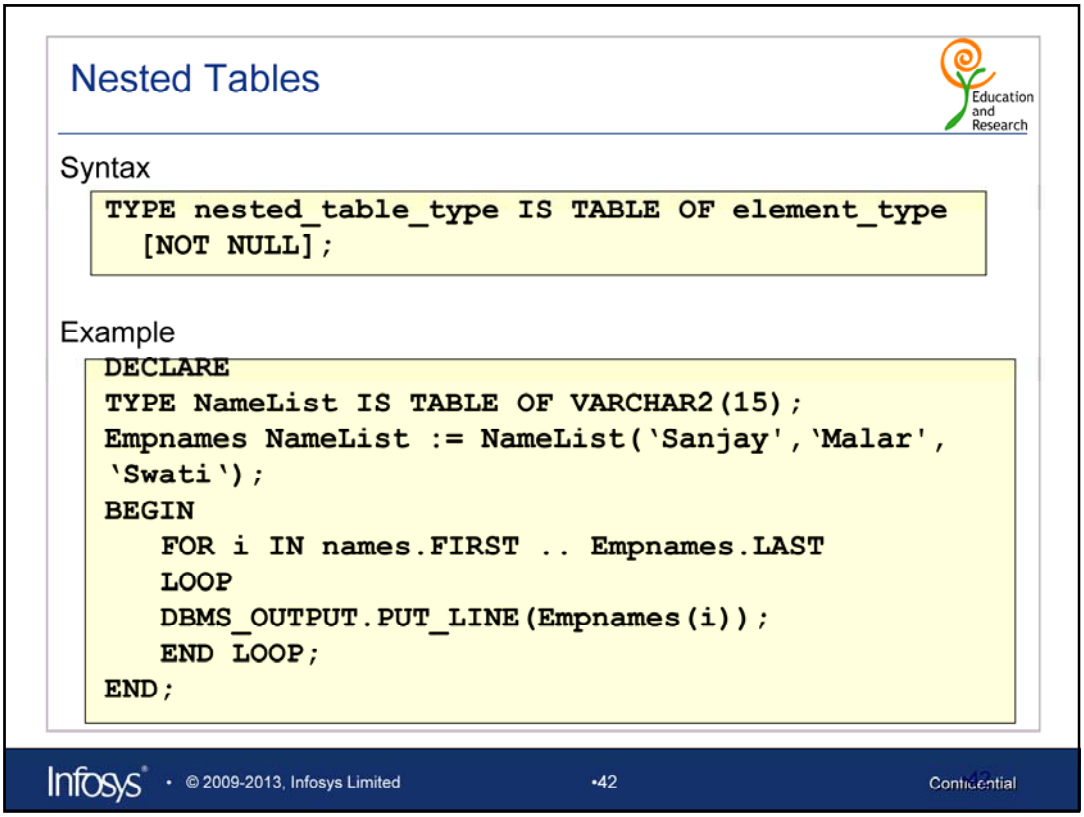
- varray_type represents any valid name that will be used for declaring a collection.
- VARRAY shows that you are declaring a collection of type VARRAY. You can also use VARYING ARRAY in place of VARRAY.
- (size) represents a positive integer value, which is used to set the upper bound of the elements that this collection can contain.
- element_type represents valid SQL or PL/SQL data type



Step 2:

Once the varray type is created, then varray collection has to be created using this varray type.

varray_name varray_type_name;



Nested Tables

Nested Tables are collection of elements of same data type. But they do not have an upper bound. Nested tables can be sparse. That is there can be empty element locations within the collection.

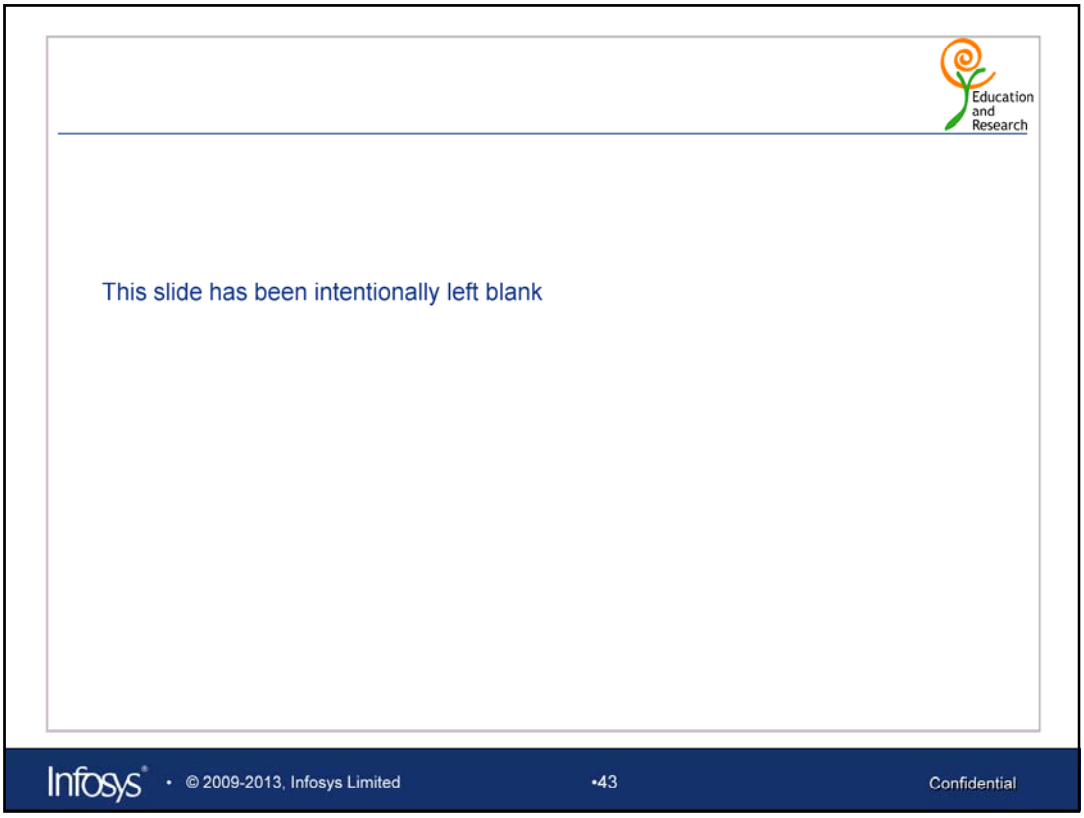
There are two steps in creating a nested table.

Step 1:

First the nested table type has to be created using the following syntax::TYPE nested_table_type IS TABLE OF element_type [NOT NULL];

In this syntax:

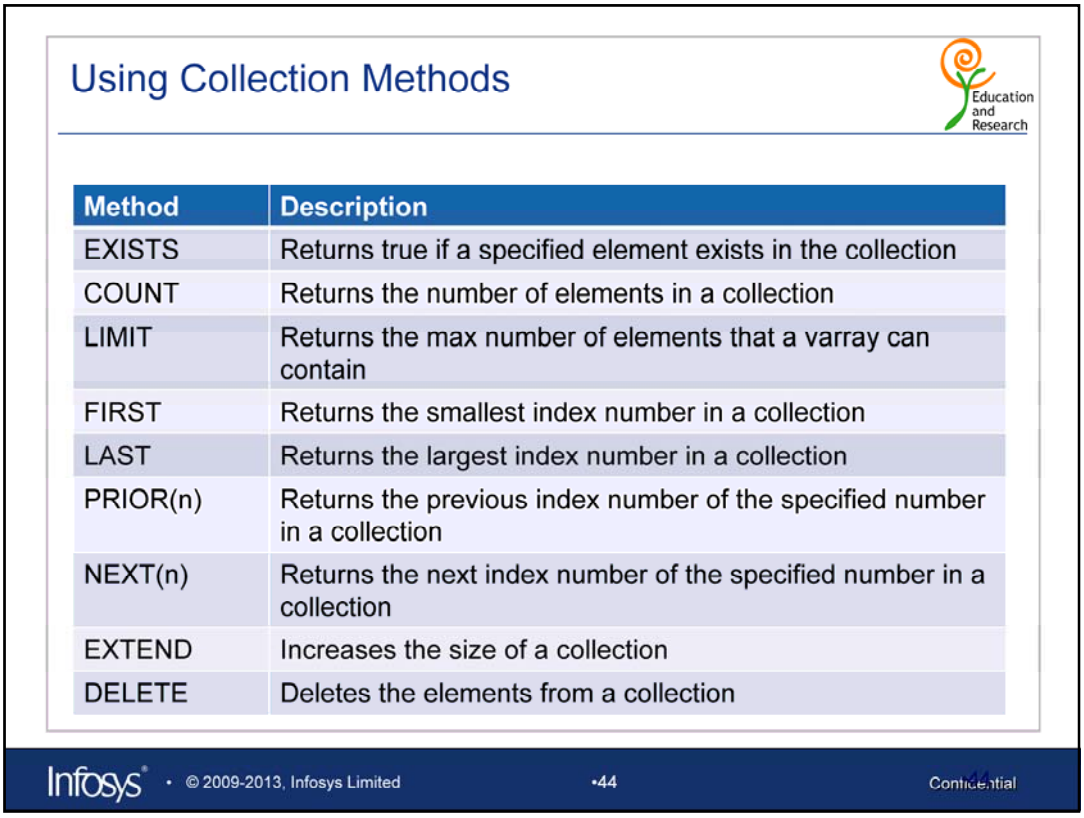
- nested_table_type is the name of the nested table type created. This will be used for declaring nested table collections
- TABLE keyword indicates that a nested table type is being created
- element_type represents valid SQL or PL/SQL data type



Step 2:

Once the nested table type is created, then nested table collection has to be created using this nested table type.

nested_table_name nested_table_type;

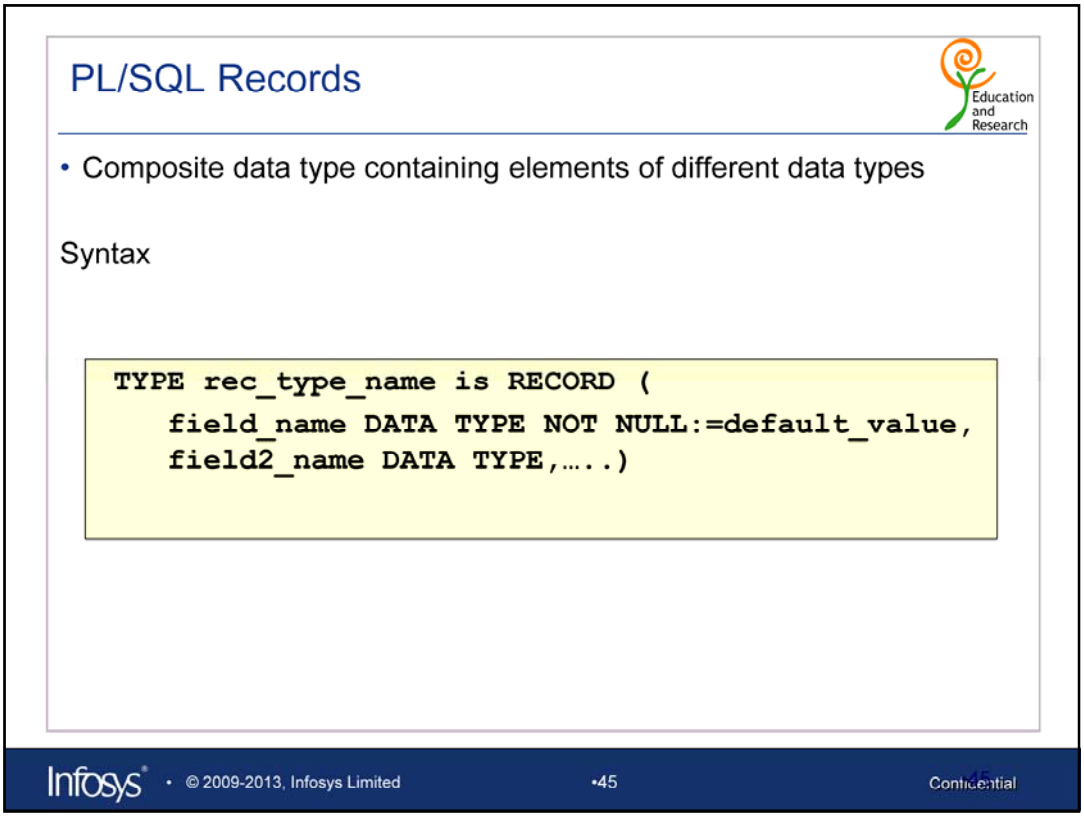


Collection Methods

Collections provide built in methods for easier application development and better maintenance. The common methods provided by collections are COUNT, DELETE, EXISTS, EXTEND, FIRST, LAST, LIMIT, NEXT, PRIOR, and TRIM.

Following points need to be considered when using the collection methods:

- These methods cannot be called from SQL statements
- COUNT, LIMIT, EXISTS, FIRST, LAST, PRIOR, and NEXT are functions
- DELETE, EXTEND and TRIM are procedures
- EXISTS, PRIOR, NEXT, TRIM, EXTEND, and DELETE take parameters corresponding to collection subscripts.



PL/SQL Records

These are collection of elements with different data types.

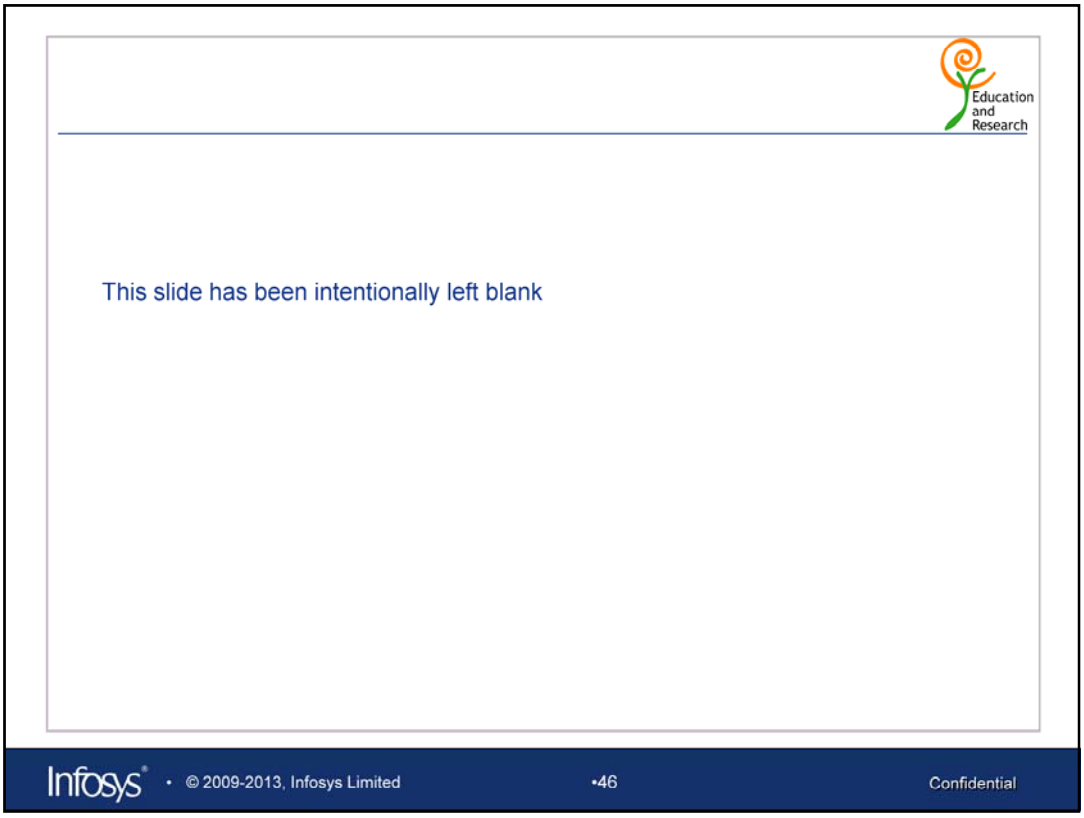
There are two steps to be followed to create a Record

Step 1:First a record type has to be created using following syntax:

```
TYPE rec_type_name is RECORD (
                field_name DATA TYPE NOT
NULL:=default_value,        field2_name DATA TYPE,…..);
```

In this syntax:

- rec_type_name represents the type name that will be used for declaring records
- field_name represents the field name of the record
- DATA TYPE represents SQL or PL/SQL data type except REF CURSOR
- NOT NULL is optional , but whenever it is used , it must be initialized with a default value.



Step 2:

Once the record type is created, then the record can be created using the record type.

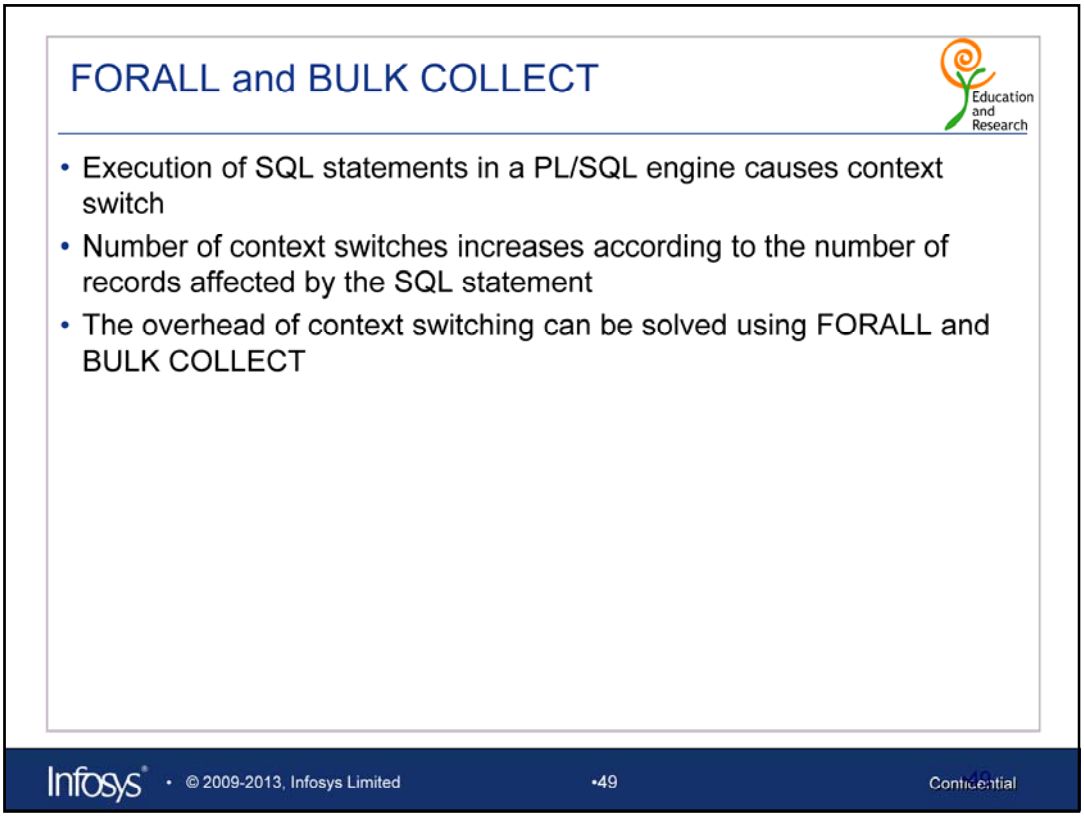
rec_name rec_type_name;

Example

```
DECLARE
    type erectype is record(
       v_eno number(4),
       v_ename varchar2(25),
       v_sal number(8,2));
  emprec emprectype;
   cursor c1 is
   select empno,empname,empsal from emp where
    empno=7567;
BEGIN
  open c1;
  fetch c1 into emprec;
  dbms_output.put_line(emprec.v_empno || '   ' ||
  emprec.v_empname || '   ' || emprec.v_sal
  close c1;
END;
```

Bulk Binding

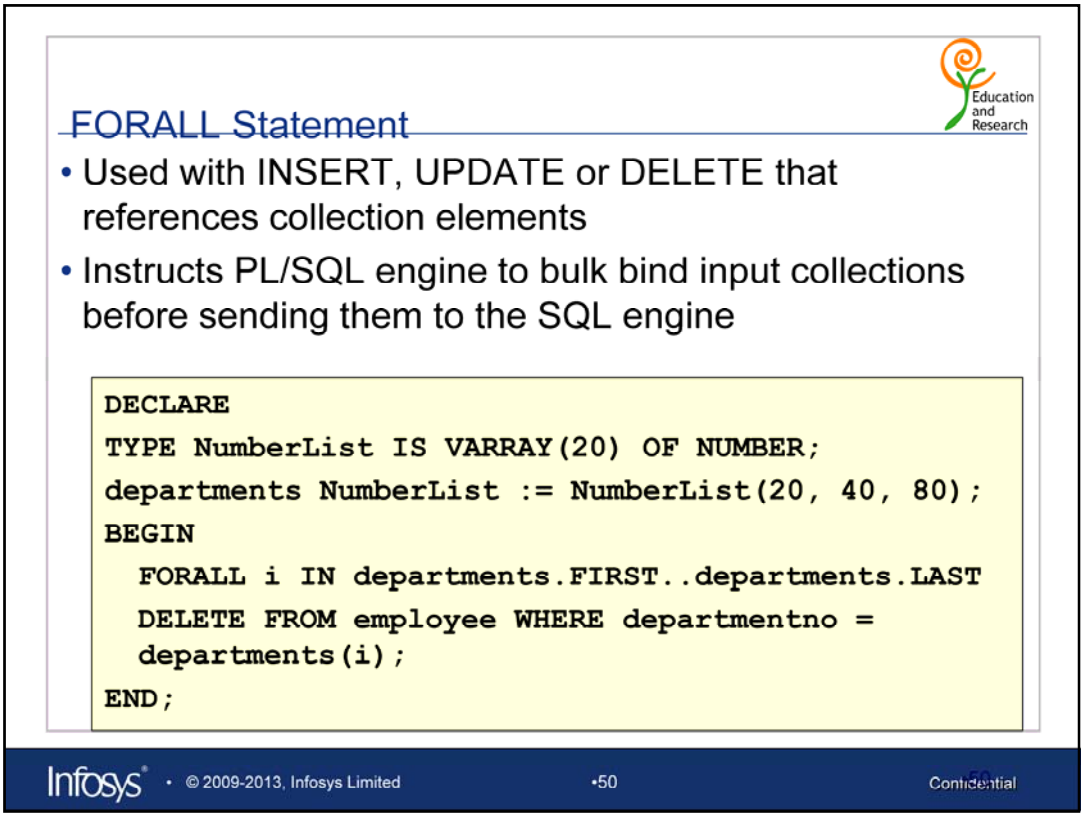
FORALL, BULK COLLECT



FORALL and BULK COLLECT

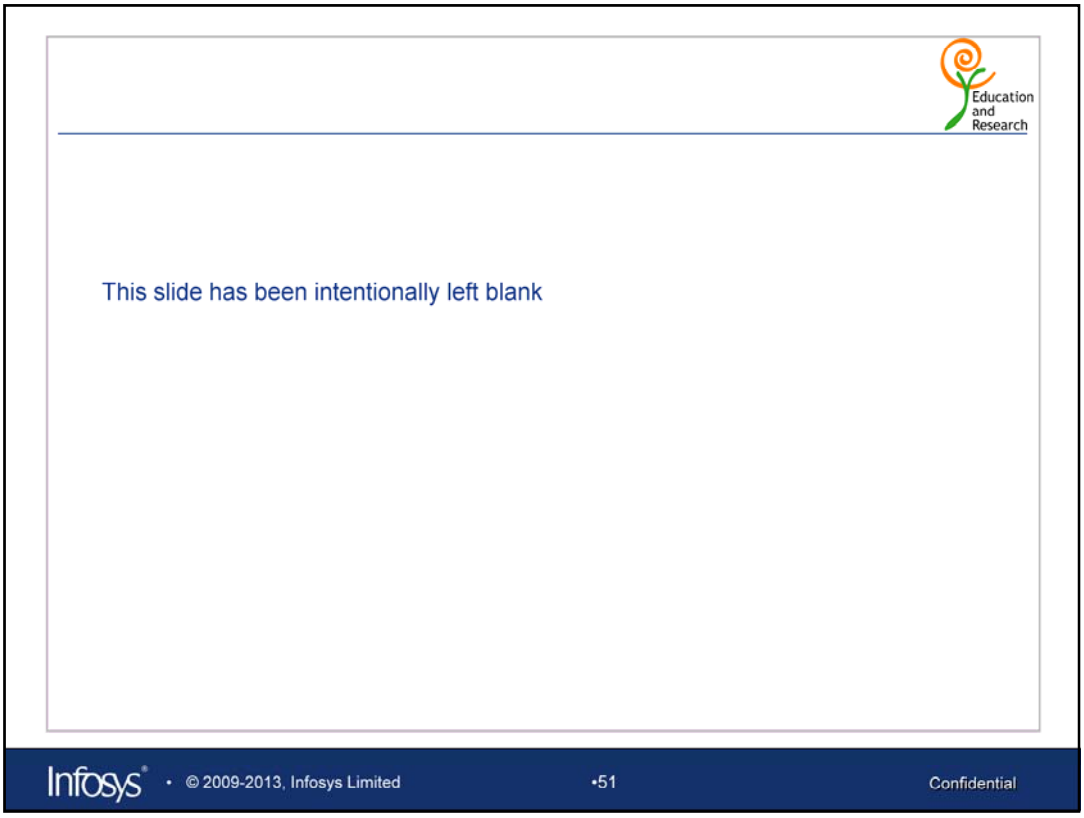
When a PL/SQL program is run, the PL/SQL statements are executed with the PL/SQL statement execution engine and the SQL statements within the program are passed over to SQL engine. This is known as context switch. The overhead of context switch degrades the performance. This problem can be solved using FORALL and BULK COLLECT statements.

FORALL and BULK COLLECT processes multiple rows of data in a single request(context switch) to the Oracle database. The SQL engine processes this request and returns all the data together (one context switch) to the PL/SQL engine, which deposits the data into one or more collections.



FORALL Statement

FORALL is used with Insert, Update or Delete statements that references collections elements. This instructs the PL/SQL engine to bulk bind the input collections, before sending them to the SQL engine. FORALL statement is not a FOR loop even though it has an iteration scheme.In the example shown in the slide, although there are three DELETE operations performed, the DELETE statement is sent only once to the SQL Engine. Thus FORALL reduces the number of context switches. FORALL statement can also be used for bulk binding a part of a collection as shown:




```
DECLARE TYPE NumberList
IS VARRAY(20) OF NUMBER;
 departments NumberList := NumberList(20,40,60,85,97,60,74,78,90,96);
BEGIN
        FORALL i IN 3..8
        UPDATE emp SET sal = sal * 1.10
        WHERE departmentno = departments(i);
 END;
```

BULK COLLECT

- Used with the SELECT statements
- Instructs the SQL engine to bulk bind the output collections before returning them to the PL/SQL engine

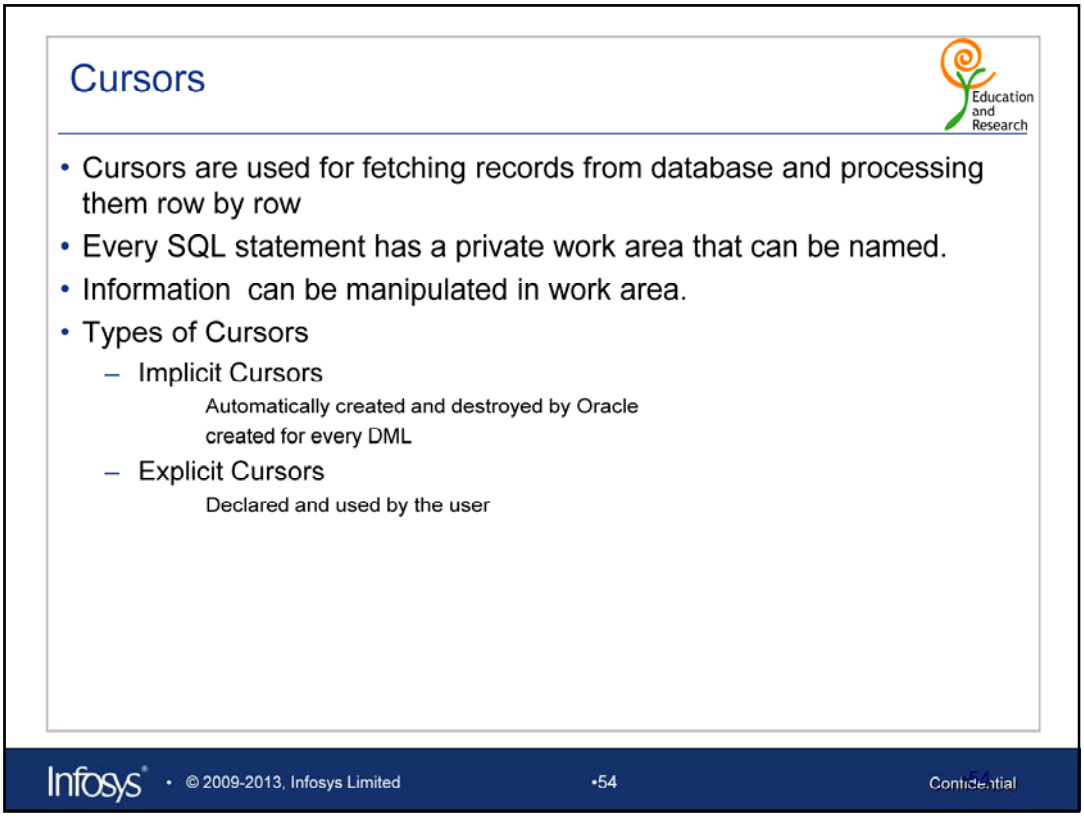
```
DECLARE
    TYPE NumberTab IS TABLE OF employee.eno%TYPE;
    TYPE NameTab IS TABLE OF employee.ename%TYPE;
    empnums NumberTab;
    empnames NameTab;
BEGIN
    SELECT eno, ename
    BULK COLLECT INTO empnums, empnames FROM
    employee;
END;
```


BULK COLLECT

BULK COLLECT is used with the SELECT statements. This clause is used to instruct the SQL engine to bulk bind the output collections before returning them to the PL/SQL engine.

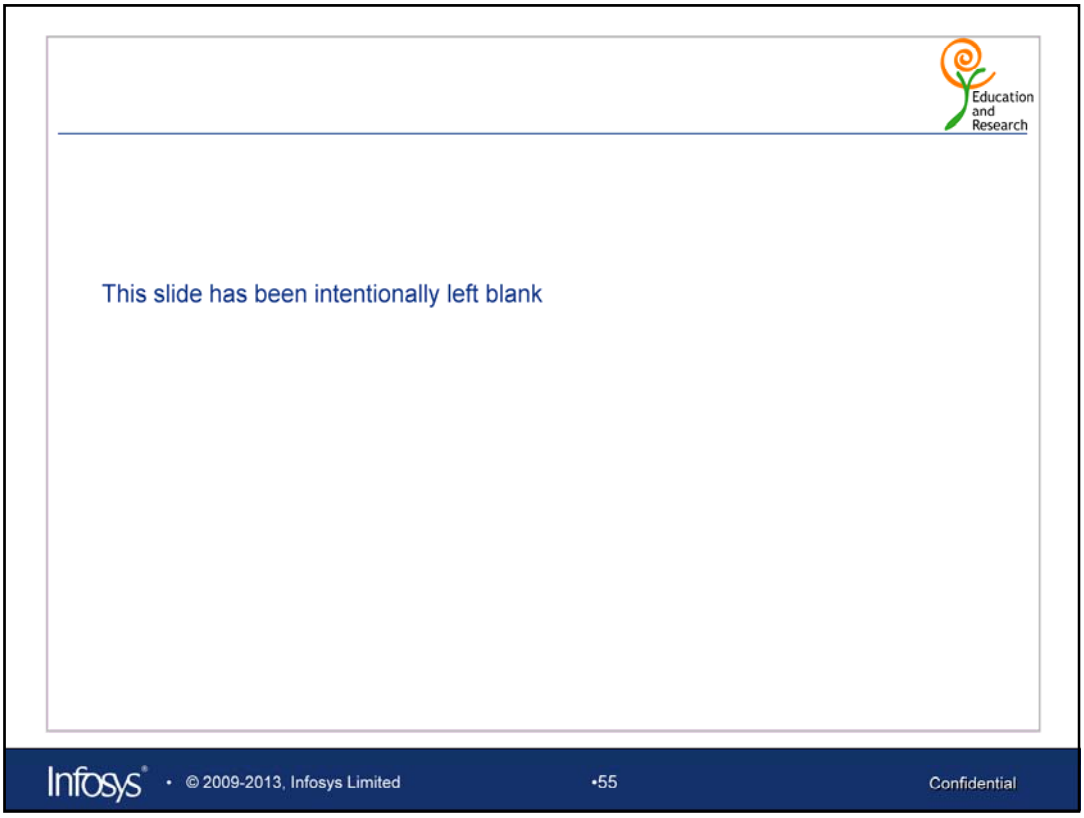
Cursors

Implicit and Explicit



Cursors

Cursor provides a technique to fetch information from database into the PL/SQL program and process them row by row.A private work area is created and assigned by Oracle, when an SQL statement from a PL/SQL program is executed. This area consists information about the SQL statement that got executed and also the data returned or that which is affected by that statement. This private work area can be named using PL/SQL cursor which in turn can be used to manipulate the information within it.Cursor gives the programmer a complete control over opening, closing and fetching the rows. Fetching from a cursor will never raise a NO_DATA_FOUND or a TOO_MANY_ROWS exception.



Types of Cursors

a) Implicit Cursors
b) Explicit Cursors

Implicit cursors are automatically created and destroyed by ORACLE. In order to reference an implicit cursor with an attribute, refer to it as SQL%<attribute_name>.

Explicit cursors are explicitly declared in the PL/SQL declare section. The attributes of the cursor can be accessed using the syntax <cursro_name>%<attricute_name>.

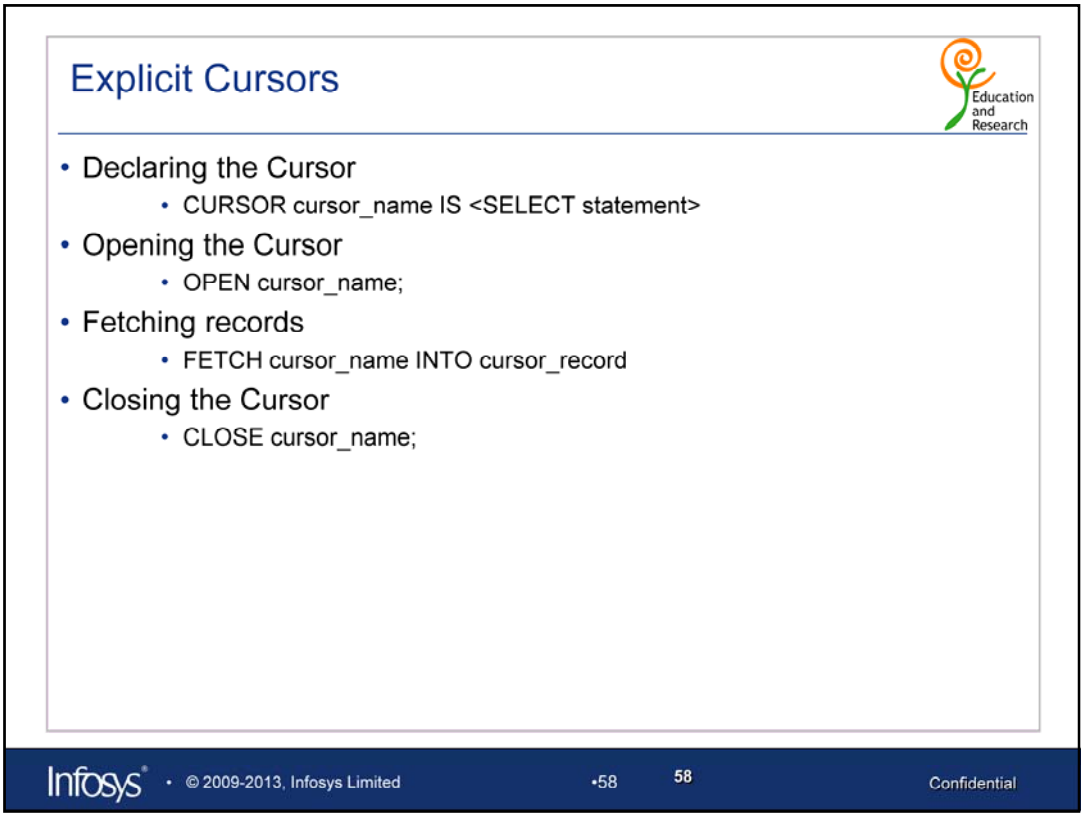
Implicit Cursors

Implicit Cursor Attributes

- SQL%FOUND
  - If one or more rows are accessed by the last DML statement, this attribute returns true
- SQL%NOTFOUND
  - Logical opposite of SQL%FOUND
- SQL%ROWCOUNT
  - The number of rows accessed by the last DML statement is returned
- SQL%ISOPEN
  - Always false as it is closed by Oracle as soon as the statement is executed

Example

```
DECLARE
  departmentnum number(4);
BEGIN
  departmentnum:=&depno;
  delete from employee where
        deptno=departmentnum;
  if (sql%found) then
   dbms_output.put_line(sql%rowcount);
  end if;
  if (sql%notfound) then
    dbms_output.put_line (departmentnum||' has no
   employees working for it');
  end if;
END;
```

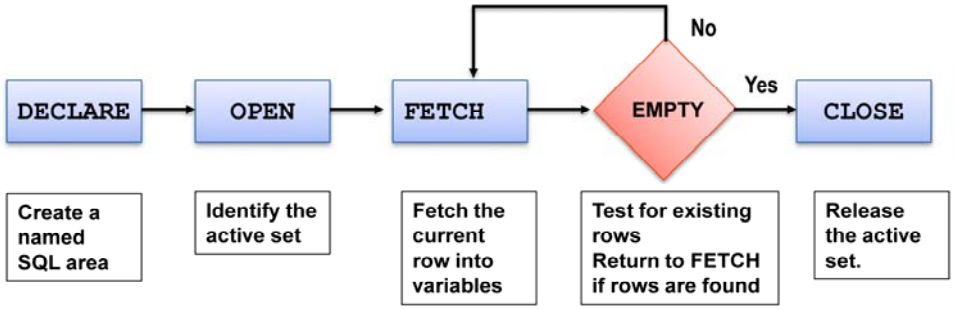


Explicit Cursors

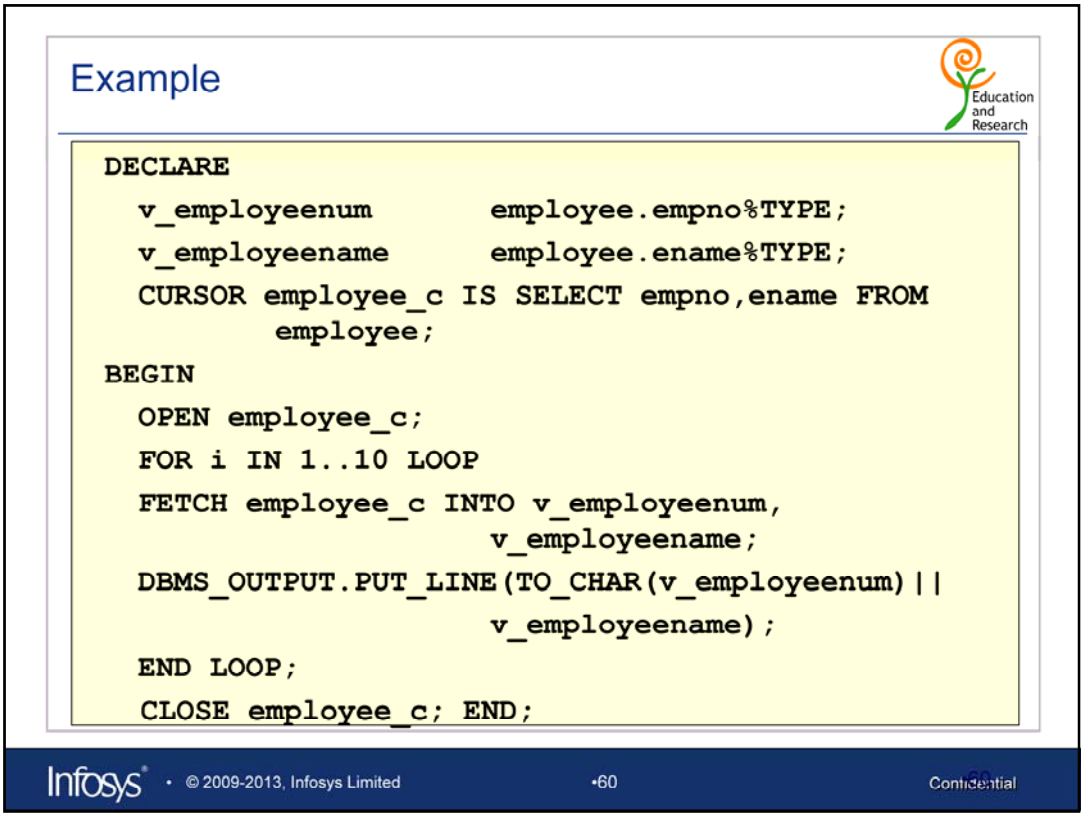
Points to remember:

- A cursor is not a PL/SQL variable, and hence cannot be part of an expression.
- Parameters can be passed to a cursor, but it cannot be used to return data from the cursor.
- The parameter which is passed to the cursor is scoped within that cursor.

Explicit Cursors

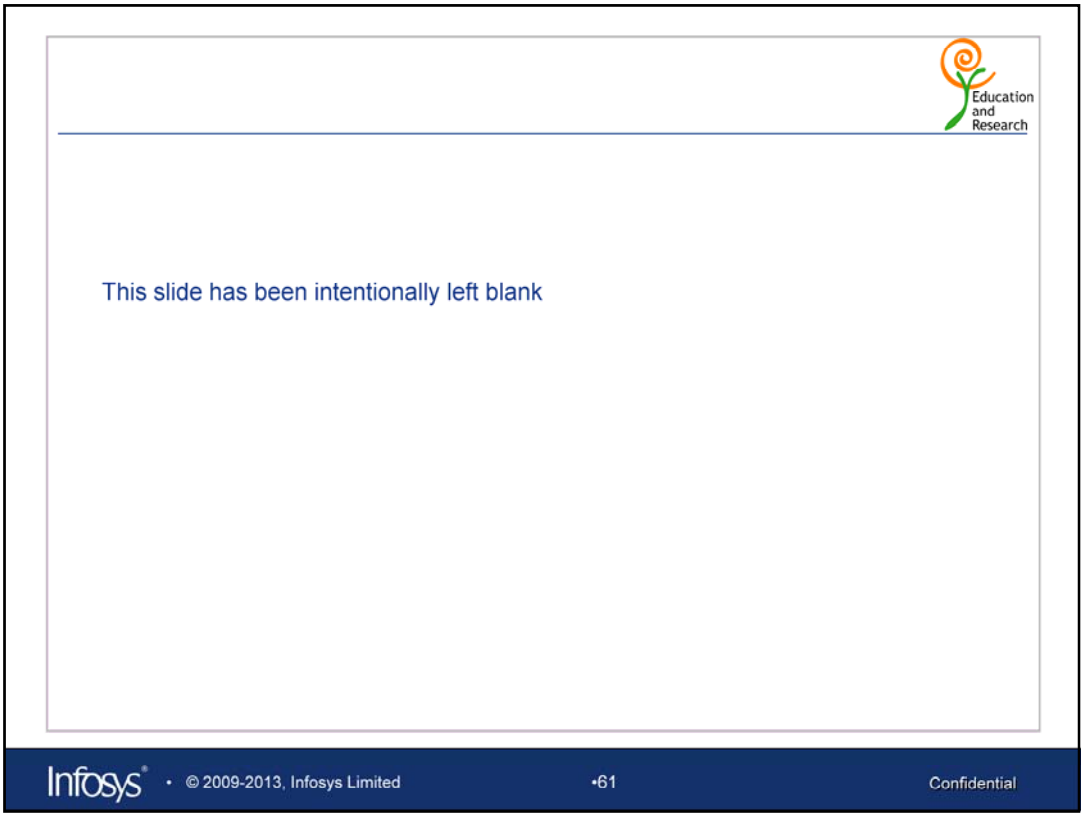


•*Source : Created by author for Infosys Technologies*

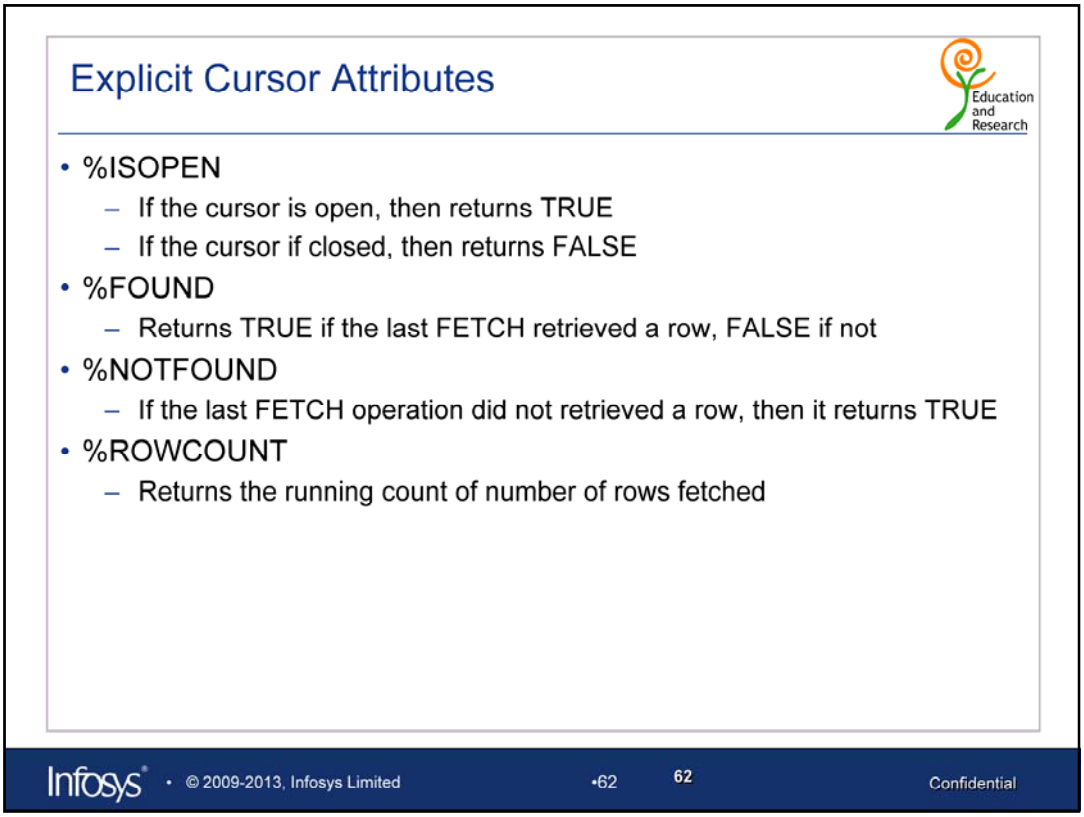


Example for Cursors

The example in the slide show how to create a cursor and associate with SELECT statement. This allows us to access individual records returned by the SELECT statement and process them.We can also achieve the same thing using sub queries as shown in the below example:




```
DECLARE
        Bonus_amt REAL;
BEGIN
        FOR employee_record IN (SELECT empno, sal, comm FROM
employee) LOOP
                Bonus_amt := (employee_record.sal * 0.05) +
                (employee_record.comm * 0.25);
                INSERT INTO bonuses VALUES
                (employee_record.empno, bonus_amt);
        END LOOP;
        COMMIT;
END;
```



Explicit Cursor Attributes

The Explicit Cursor attributes are used to get the current status of cursors. The cursor attributes are,

%FOUND

Returns TRUE if the last FETCH retrieved a row, FALSE if not

%NOTFOUND

If the last FETCH operation did not retrieved a row, then it returns TRUE

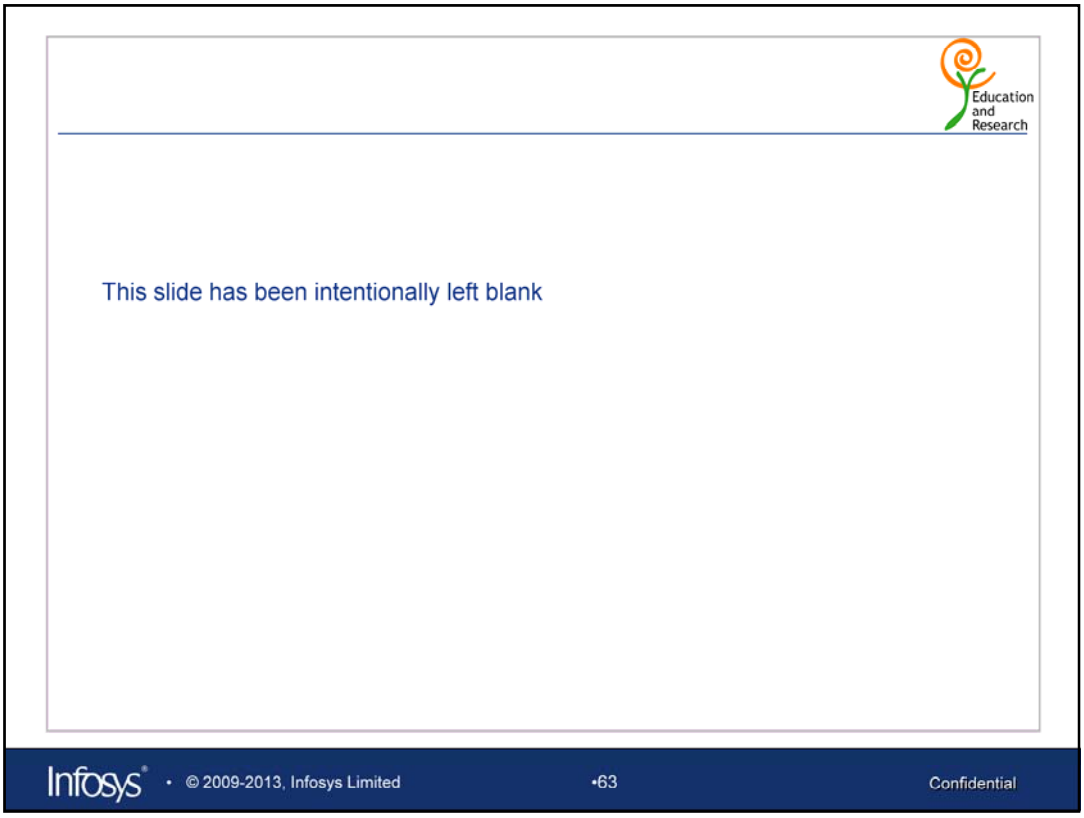
%ROWCOUNT

Returns the running count of number of rows fetched

%ISOPEN

If the cursor is open, then returns TRUE

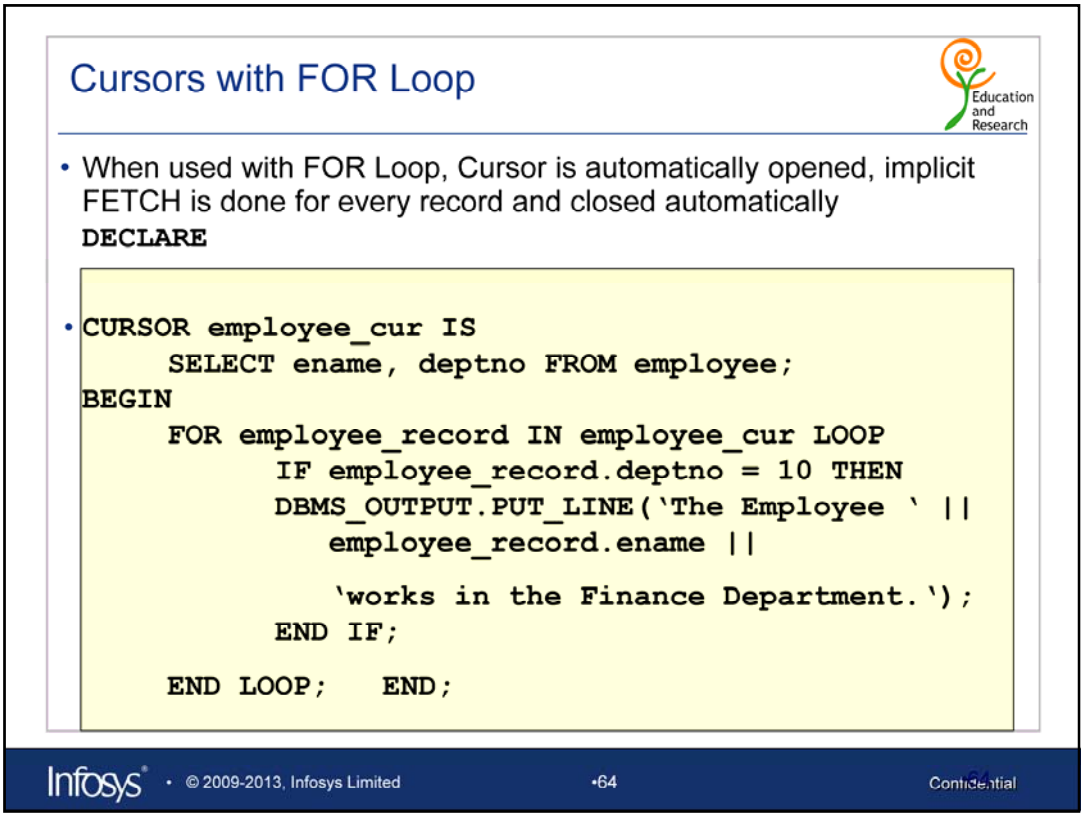
If the cursor if closed, then returns FALSE



Example:

```
BEGIN
OPEN Employee_cur;
LOOP
        FETCH Employee_cur INTO Emploee_record;
        EXIT WHEN Employee_cur%ROWCOUNT > 10 OR
        Employee_cur%NOTFOUND:
END LOOP;
CLOSE Employee_cur;
```

In the example above the loop is terminated when 10 records are fetched from the cursor or it reaches to the end of cursor.

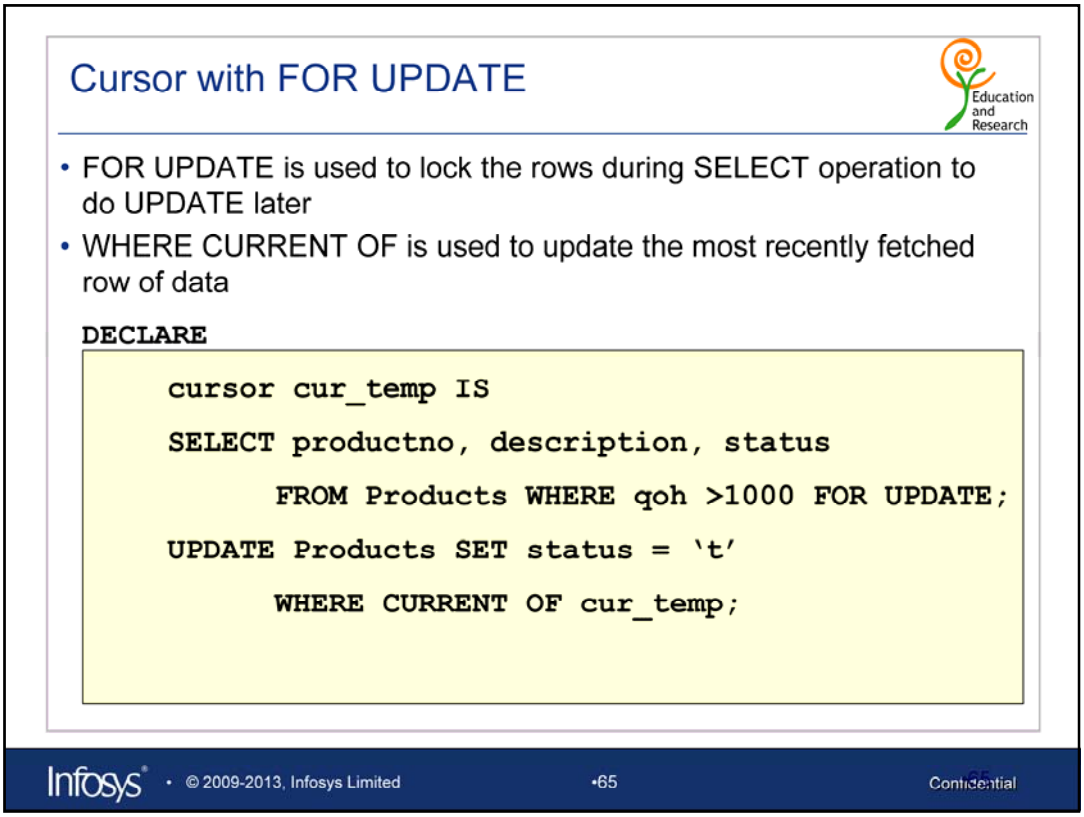


FOR Loops in Cursor

When a FOR loop is used with the cursor, the cursor is automatically opened when the loop gets initiated. The loop iterates for every record returned by the SELECT statement associated with the cursor and does an implicit FETCH operation. Once all the row are fetched as per the query, then the cursor is automatically closed and the loop terminates. The other situations when a cursor gets closed automatically are

-An Exception is raised inside the loop

-An EXIT or GOTO statement is used to leave the loop



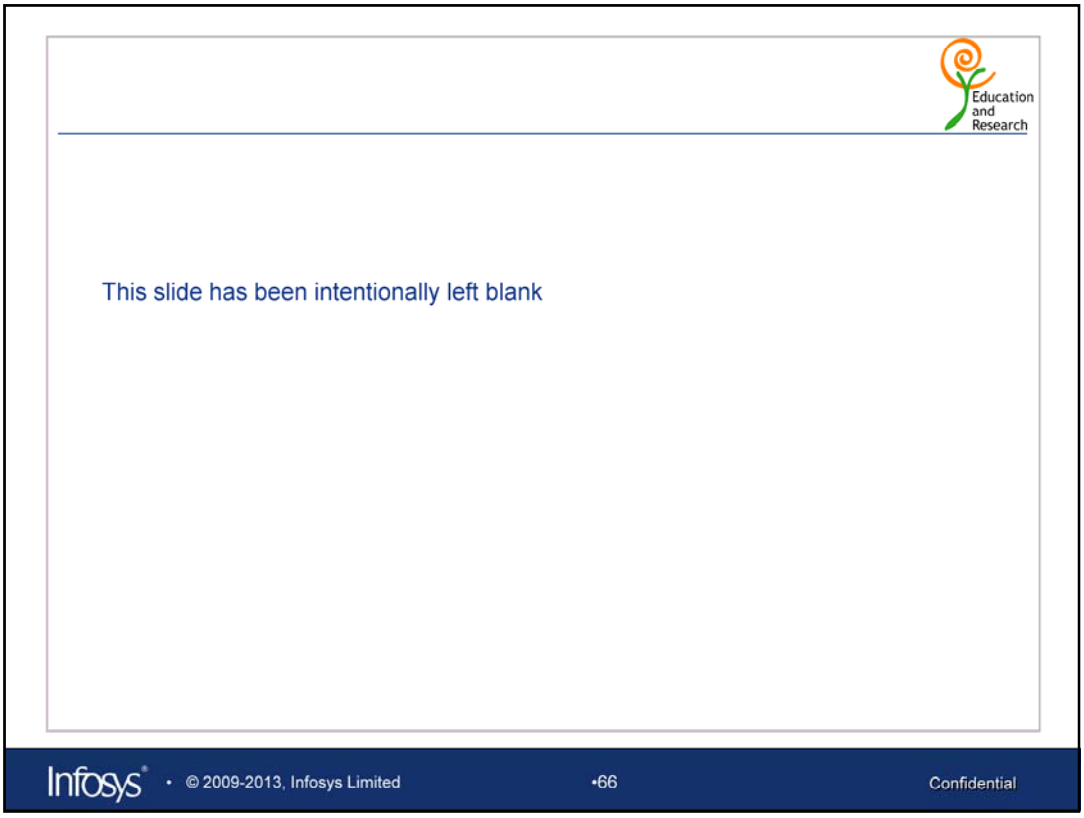
Using the For Update of With Cursor

Whenever a SELECT statement is executed to query database for a set of records, it doesn't take an exclusive lock. But sometimes, we may want to take an exclusive lock on a set of records, even before we change them. FOR UPDATE clause provided by Oracle is used to do this.Example:

```
DECLARE CURSOR temp_cur IS
        SELECT product_name, manufacturer FROM Products
                WHERE manufacturere = 'ABC'
                FOR UPDATE;
```

Points to remember:

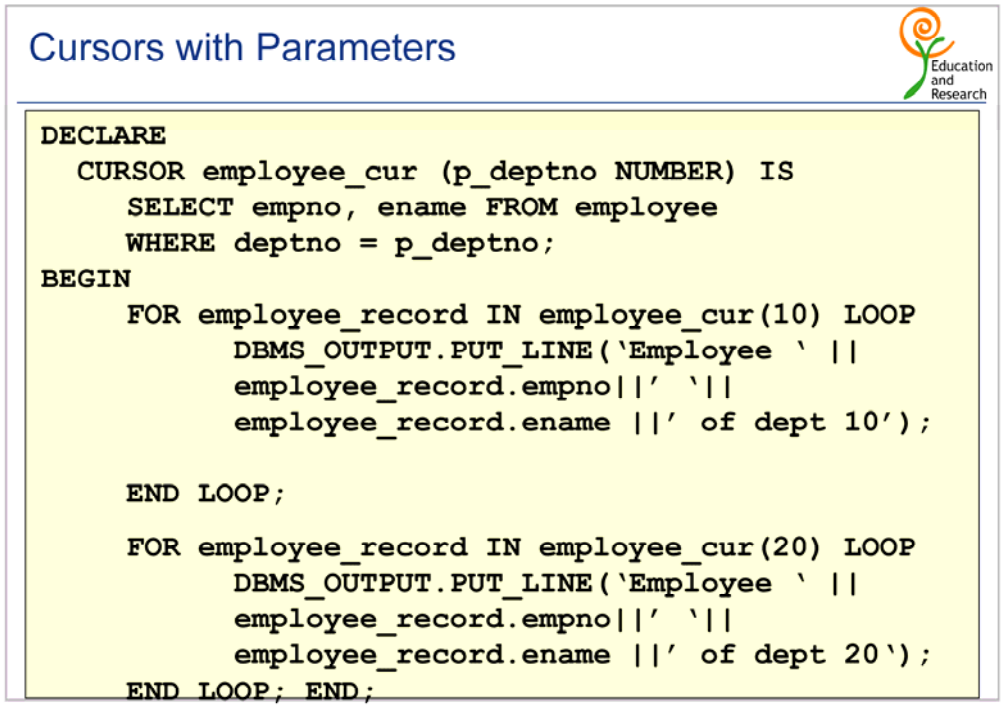
- You can qualify the for update of clause with a column name to indicate explicitly which column is to be updated.
- Locking does not restrict you to update other columns which are not part of the SELECT statement.



Once the cursor is opened, locks are put on the specified result set. Using a commit or rollback releases all locks on a table. So place the commit or rollback after some processing logic or after all operations with the cursor are completed.

The WHERE CURRENT OF clause is used to make changes to the most recently fetched row of data. This clause can be used with both the update and delete statements.

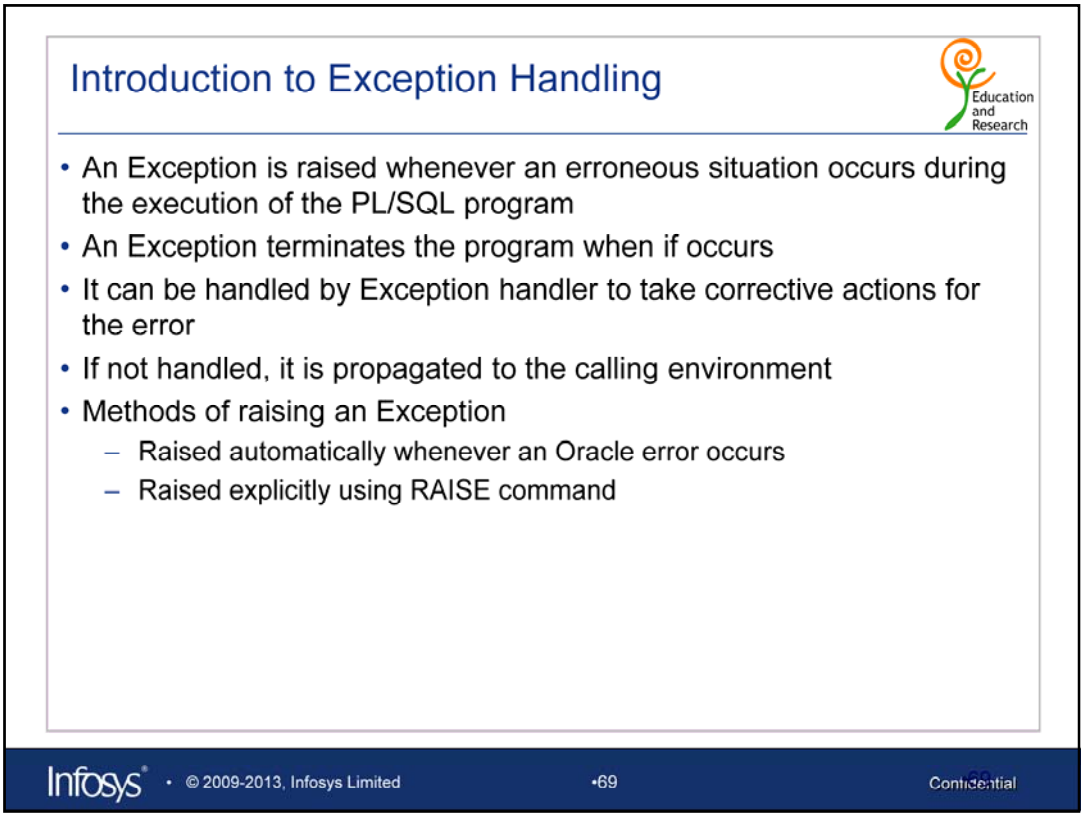
Cursors with Parameters



Passing Arguments to a Cursor

You can declare a cursor to receive arguments. The arguments passed to cursor can then be used as part of the select statement, to evaluate the cursor each time the code is executed with different set of values, as cursors, come into existence only when the cursor is opened.

Exception Handling



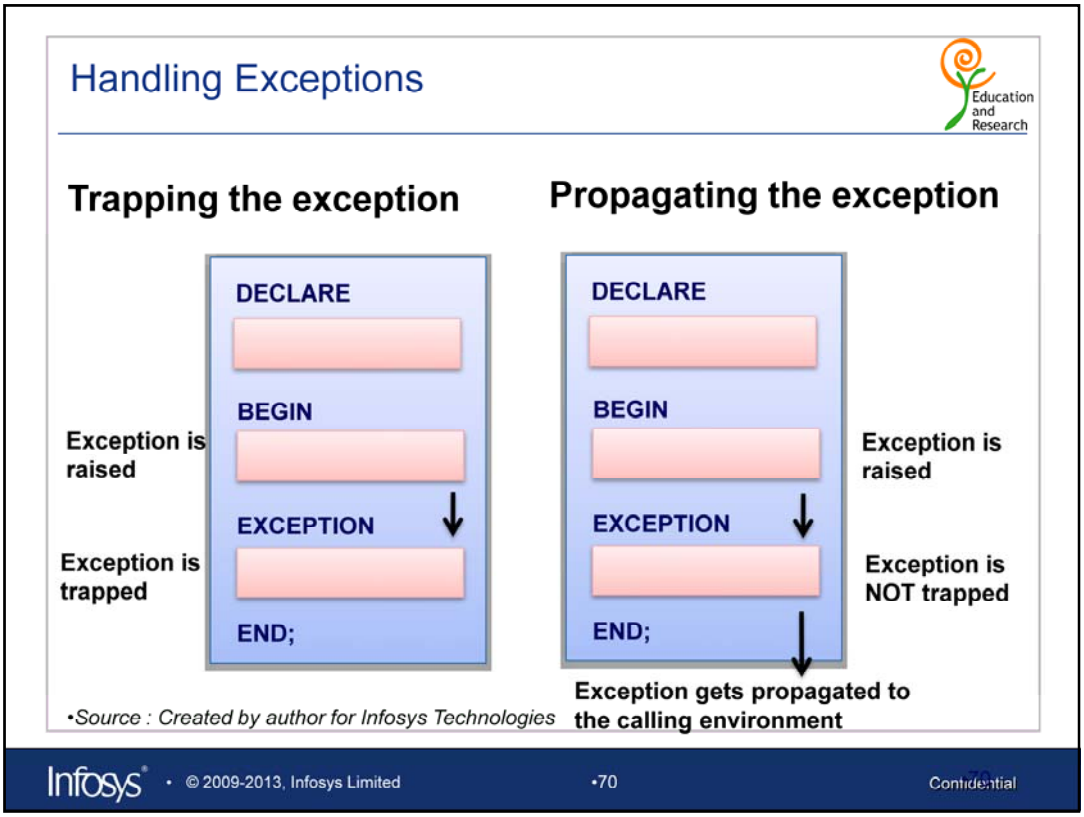
Exception Handling

An exception is raised, whenever there is an erroneous situations during the execution of the PL/SQL program. The exception is declared as an identifier in the PL/SQL program. The program terminates when an exception is raised. But the exception which is raised can be handled by set of statements within the PL/SQL program. This set of statements is called the Exception handler. The exception handler performs the final set of actions to be performed, before the program terminates. But if the exception is not handled, then it gets propagated to the calling environment.

Methods of raising an Exception

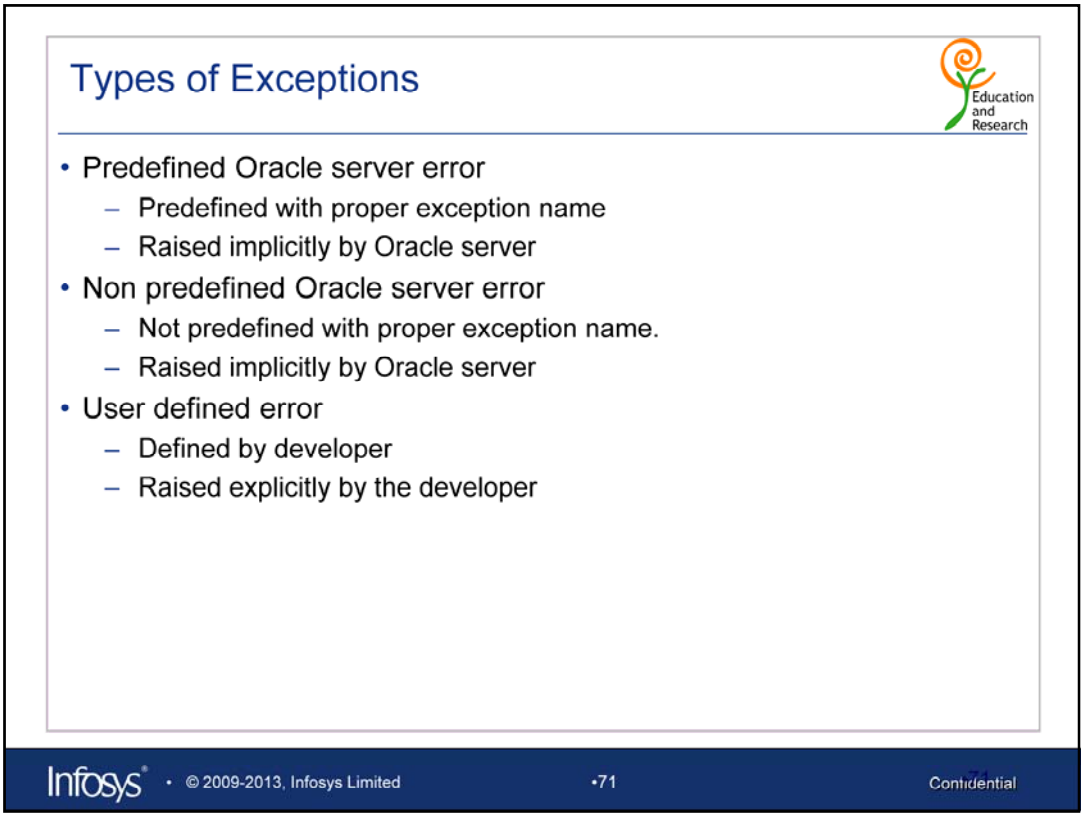
a) An exception is raised automatically whenever an Oracle error occurs. For example, if a SELECT statement is unable to retrieve any records, then an Oracle error ORA-01403 occurs. When this error occurs, PL/SQL raises an exception called NO_DATA_FOUND exception.

b) An exception can also be raised explicitly by issuing RAISE command within the PL/SQL program. This can be a predefined exception or an user-defined exception



Trapping an Exception

An exception is normally raised in the executable section of the PL/SQL block and this is handled by the corresponding exception handler in the exception section of the block. If the exception is handled successfully, then the PL/SQL block terminates with success. But if an exception handler for a particular exception is not present, then the PL/SQL block terminates with failure and the exception gets propagated to the calling environment.



Types of Exceptions

There are three types of exceptions:

1) Predefined Oracle Server Error

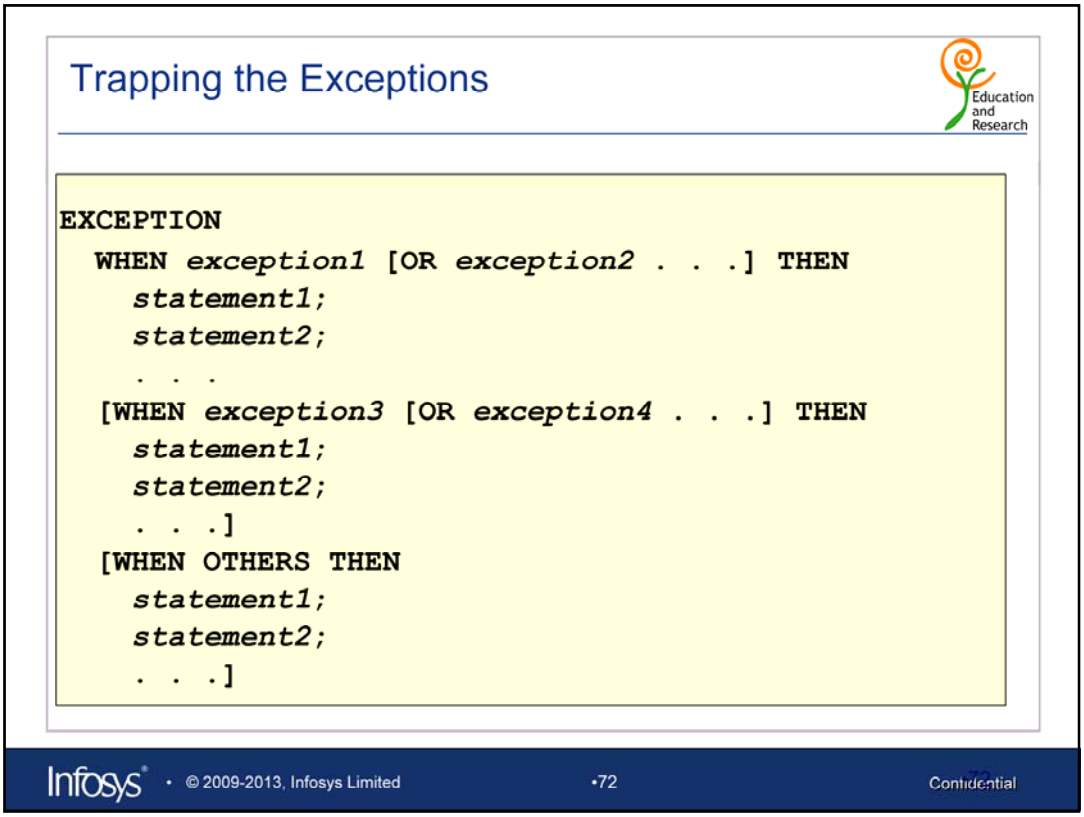
   These are predefined in Oracle server and are raised implicitly.

2) Non-predefined Oracle Server Error

   These are Oracle server error which are also raised implicitly, but these are not named by the Oracle server. They just have error code, which can be assigned with an exception name using the PRAGMA INIT declarative.

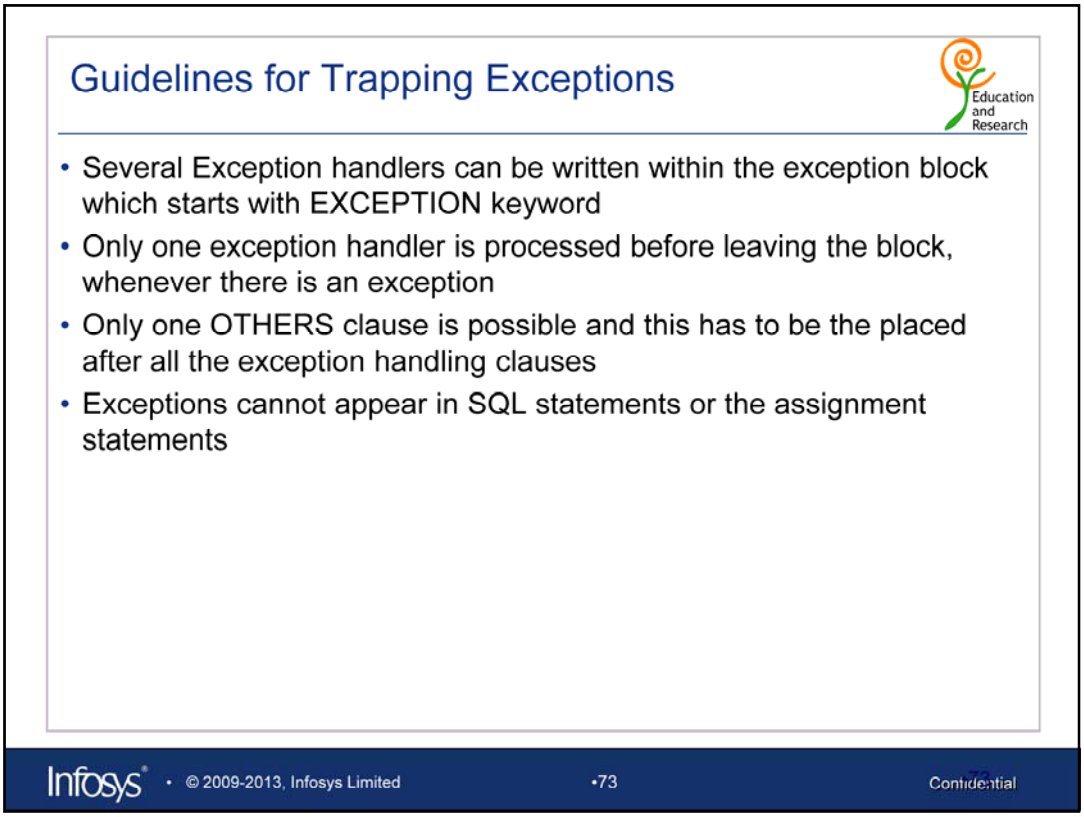
3) User defined Error

   These are defined by the developer and raised explicitly in the PL/SQL block



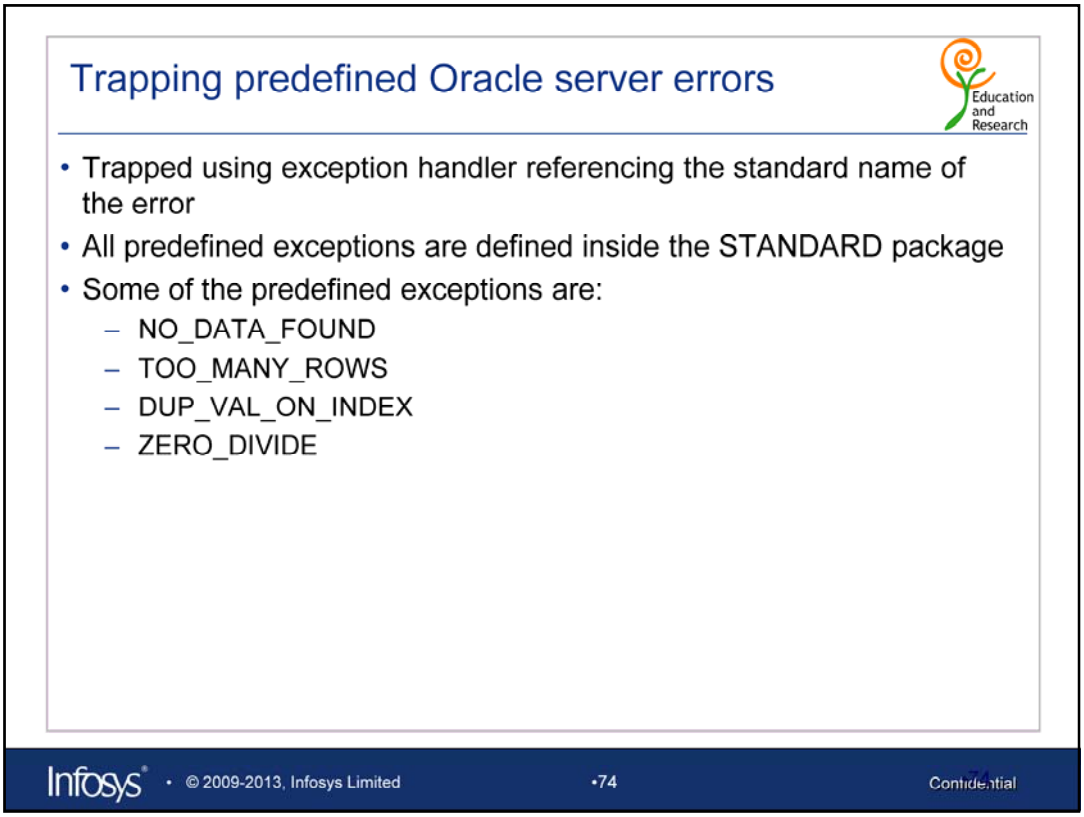
Trapping the Exceptions

An exception is raised from the executable section of the PL/SQL block. This exception can be trapped and handled by an exception handler within the exception section of the PL/SQL block. An exception handler is written using a WHEN clause followed by a sequence of statements to be executed to handle the exception raised.Those exceptions which are raised in the block but do not have a exception handler defined to handle them, can be handled using the clause WHEN OTHERS. This traps any exception that is not handled and hence this should be the last exception handler defined in the exception section.



Guidelines for Trapping Exceptions

a) Define exception handlers for each possible exception that may be raised with the PL/SQL code. All the Exception handlers must be defined within the exception block which starts with a keyword EXCEPTION
b) PL/SQL processes only one exception handler before leaving the block, whenever there is an exception
c) There can be only one OTHERS clause and place this after all the exception handling clauses
d) WHEN OTHERS is an optional clause
e) Exceptions cannot appear in the SQL statements or assignment statements

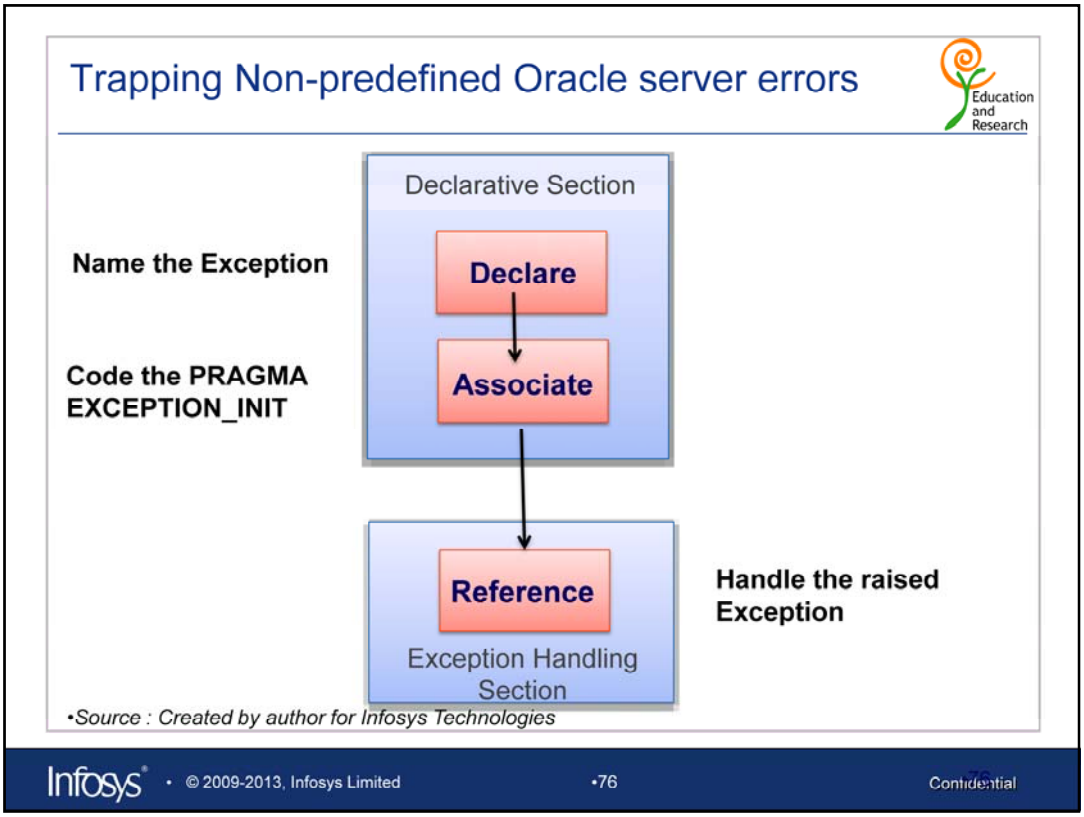


Trapping predefined Oracle server errors

The predefined Oracle server errors can be trapped using an exception handler referencing the standard name of the error. All the predefined exceptions are declared in the STANDARD package. The most common predefined exceptions which occur very frequently are NO_DATA_FOUND and TOO_MANY_ROWS exceptions. Hence it is a best practice to handle these exceptions in all the PL/SQL blocks.The normal execution of the PL/SQL code is stopped when an exception is raised. When the exception is trapped by the corresponding exception handler, the control is transferred to the exception handling section and the corrective action defined inside the exception handling section is executed. But after execution of the exception handler, the control cannot not go back to the executable section to resume the processing where it left off. The block will be terminated after successful execution of the exception handler.

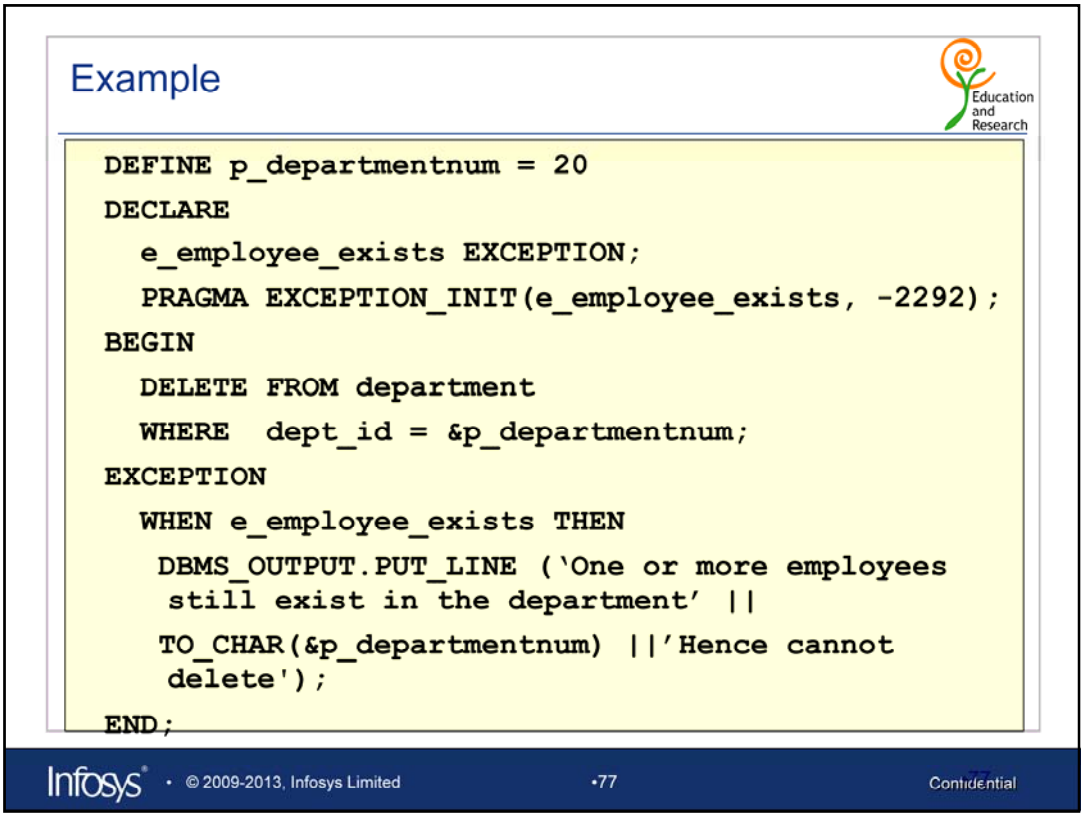
Example

```
DECLARE
   Price_Earnings number(4,2);
BEGIN
   SELECT price/earnings INTO Price_Earnings
         FROM stocks WHERE symbol ='abc'
   INSERT INTO statistics(symbol,ratio)
   VALUES('xyz',pe_ratio);
EXCEPTION
WHEN ZERO_DIVIDE THEN
   INSERT INTO statistics(symbol,ratio) VALUES
  ('abc',NULL);
WHEN OTHERS THEN  ROLLBACK;  END;
```



Trapping Non-predefined Oracle server errors

The Non-predefined Oracle server errors have only an error code to identify. They do not have a exception name to identify and handle. Hence to assign a name to the non-predefined oracle server error, a compiler directive called PRAGMA EXCEPTION_INIT is used, which instructs the compiler to associate the give exception name to the error code. Once the association is done, an exception handler for that exception can be written to handle the non-predefined exception.



Steps to handle the non-predefined Oracle server error

1) Declare an exception

   The exception is declared in the declarative section of the PL/SQL block. The syntax for declaring the exception is:

   <exception_name> EXCEPTION;

2) Associate with the Oracle server error

   The declared exception is associated with a Oracle server error using the PRAGMA EXCEPTION_INIT directive. The syntax for doing this is:

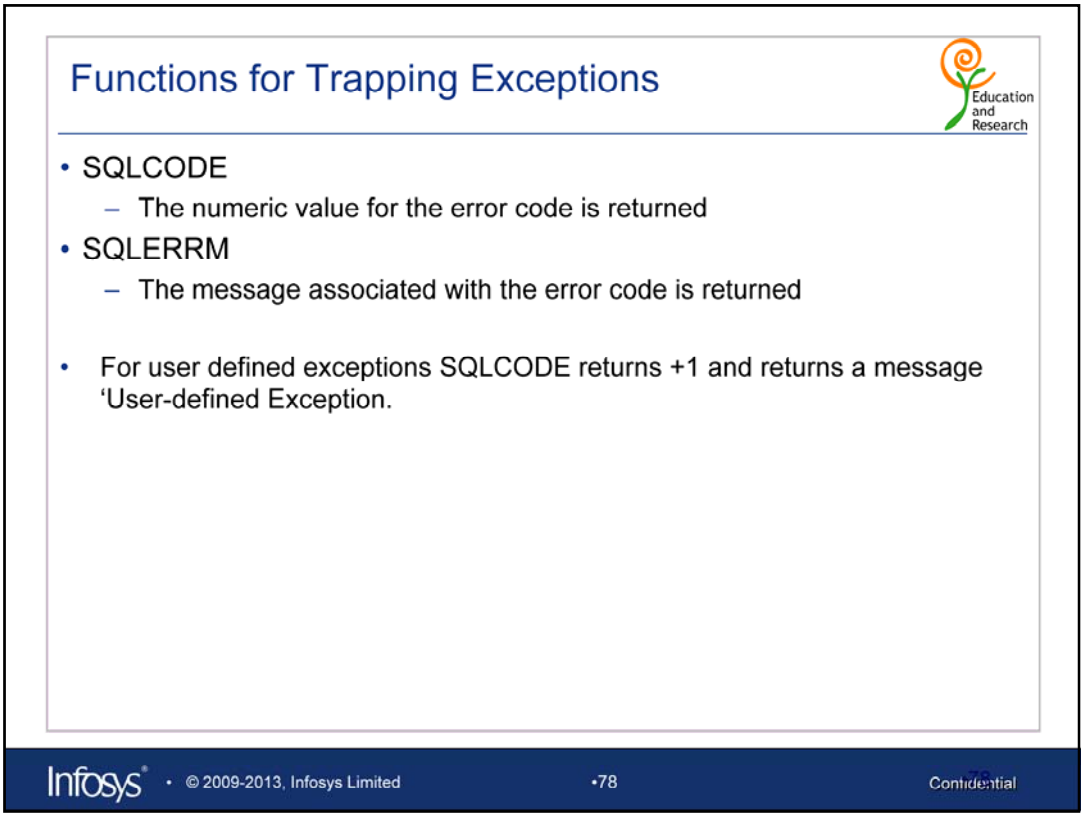
   PRAGMA EXCEPTION_INIT(<exception_name>, <error_number>);

3) Handle the raised exception

   The non-predefined oracle server exception is raised implicitly and can be handled in the exception section using the following syntax:

   WHEN <exception_name> THEN

   <statements >



Functions for Trapping Exceptions

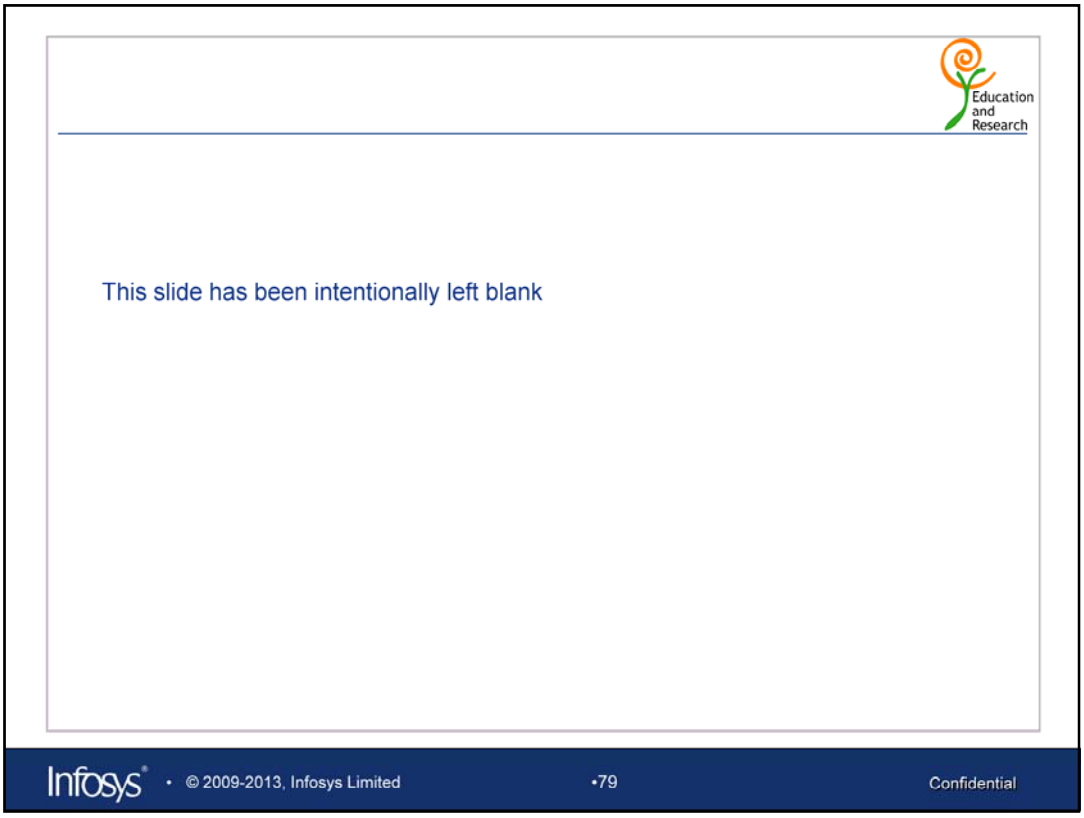
The error code and the associated message can be identified using the following two functions:

a) SQLCODE

The numeric value for the error code is returned

b) SQLERRM

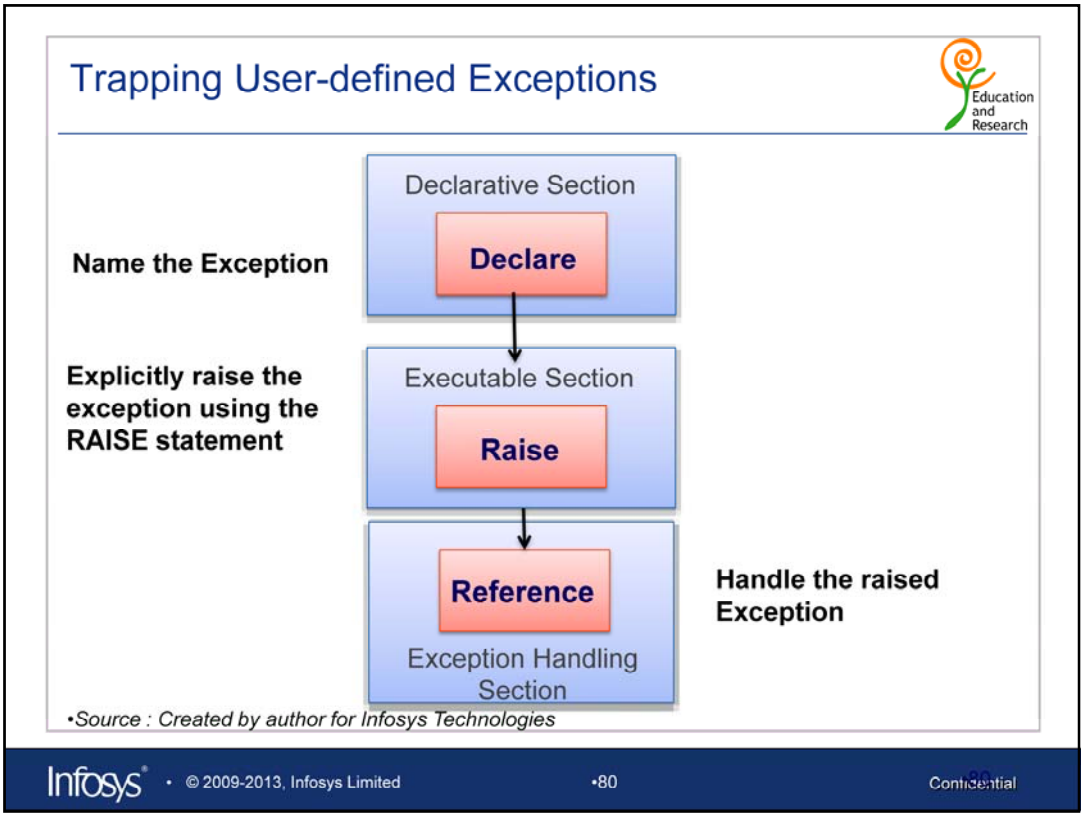
The message associated with the error code is returned



Example:

```
DECLARE
  error_number NUMBER;
  error_message VARCHAR2(50);
BEGIN
  ...
EXCEPTION
  ...
  WHEN OTHERS THEN
    error_number := SQLCODE;
    error_message := SQLERRM;
    INSERT INTO log_errors VALUES (error_number, error_message);
END;
```

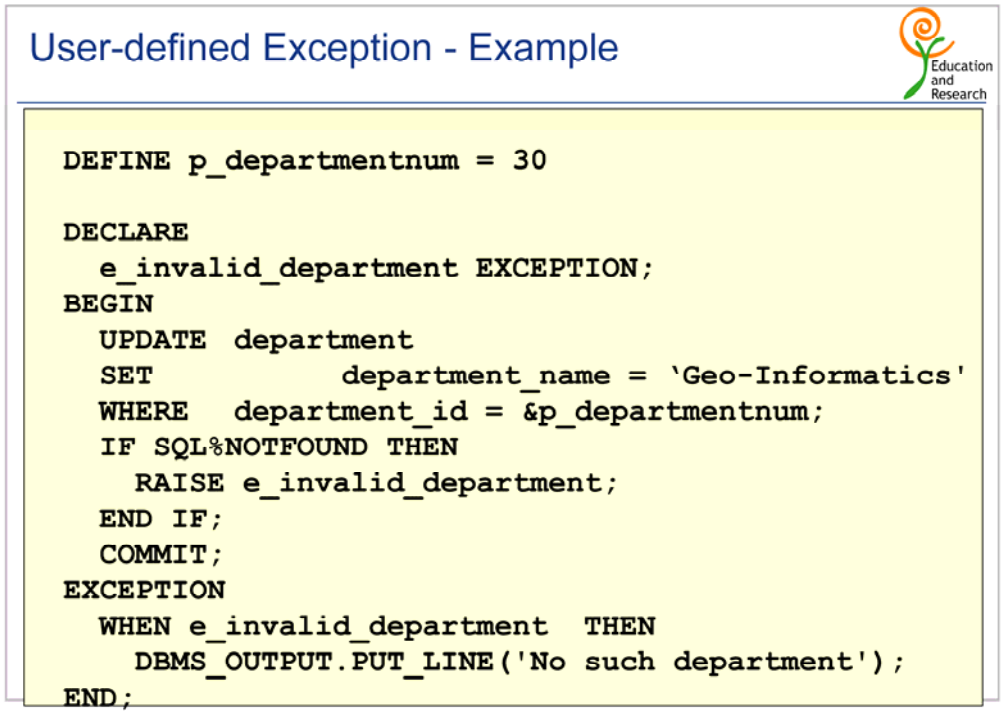
As shown above, the error code and error message have to be first stored in a local variable and then only can be inserted into table. They cannot be directly invoked in the VALUES clause to insert the error code and error message.



Trapping User-defined Exceptions

User-defined exceptions are declared in the declare section of the PL/SQL block and has to be raised explicitly using the RAISE statement. Once the exception is raised, it can be handled in the exception section of the block using the WHEN clause.

User-defined Exception - Example



```
DEFINE p_departmentnum = 30

DECLARE
  e_invalid_department EXCEPTION;
BEGIN
  UPDATE  department
  SET           department_name = 'Geo-Informatics'
  WHERE   department_id = &p_departmentnum;
  IF SQL%NOTFOUND THEN
    RAISE e_invalid_department;
  END IF;
  COMMIT;
EXCEPTION
  WHEN e_invalid_department  THEN
    DBMS_OUTPUT.PUT_LINE('No such department');
END;
```


Steps to handle the User-defined Exception

1) Declare an exception

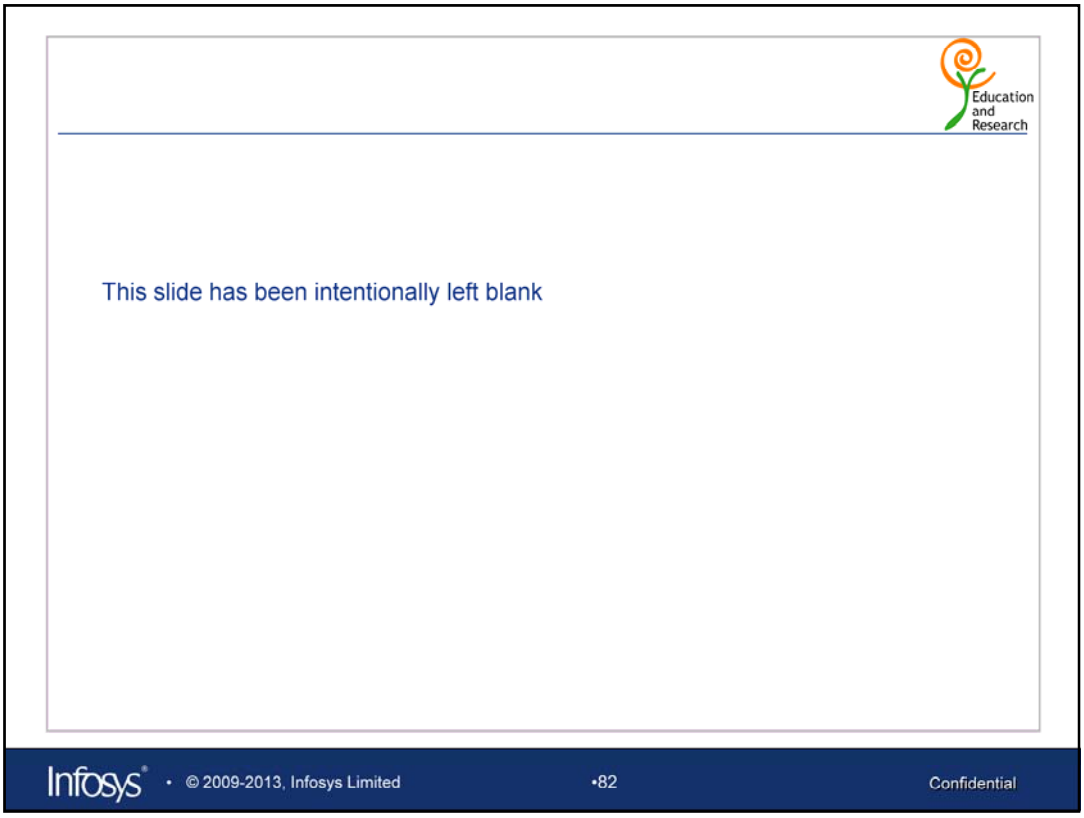
The exception is declared in the declarative section of the PL/SQL block. The syntax for declaring the exception is:

<exception_name> EXCEPTION;

2) Raise the exception

The user-defined exception is raised using the RAISE statement with the following syntax:

RAISE <exception_name>



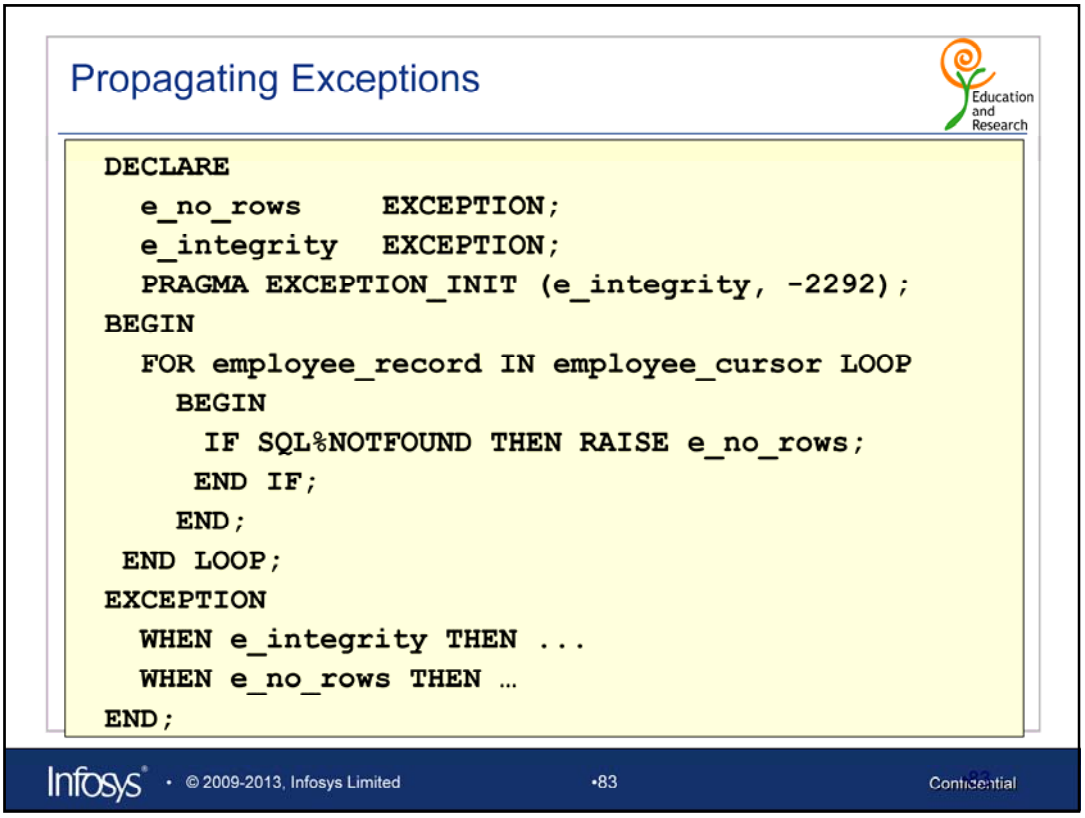
1) Handle the raised exception

The non-predefined oracle server exception is raised implicitly and can be handled in the exception section using the following syntax:

WHEN <exception_name> THEN

<statements >

The RAISE statement can also be used within the exception section inside the exception handler to raise the same exception back to the calling environment.

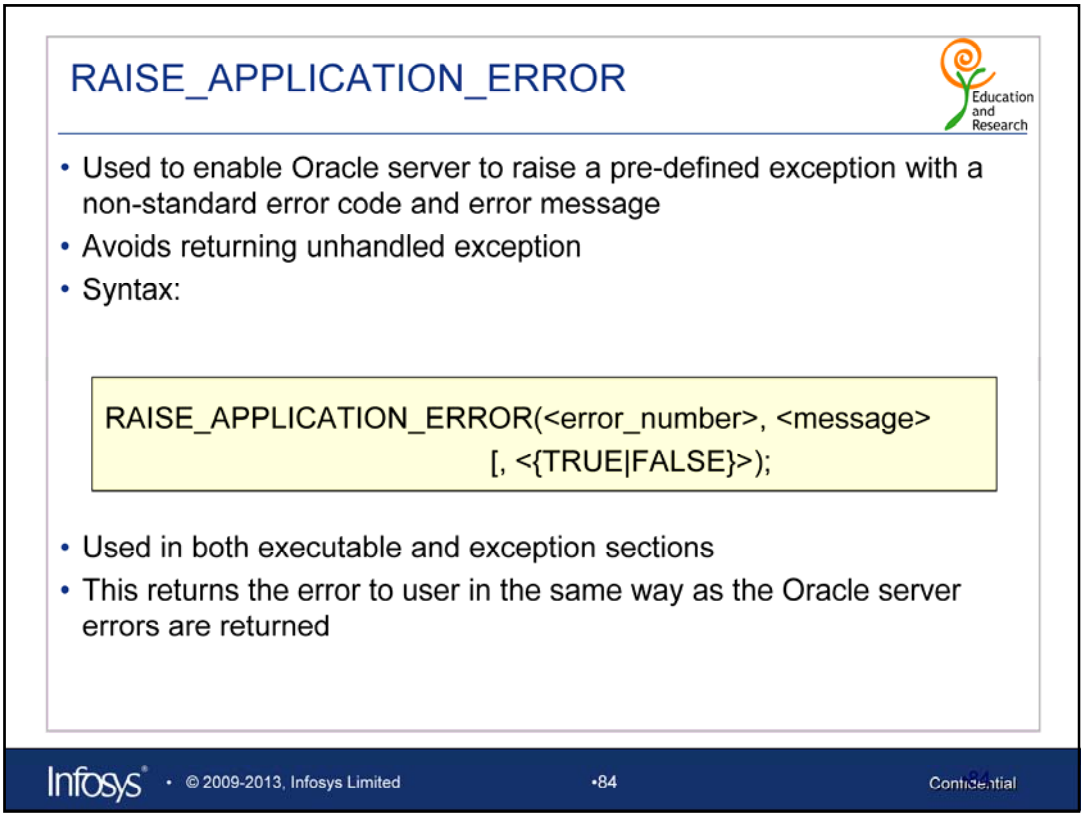


Propagating Exceptions

The exceptions, if not handled, can be propagated to the enclosing block or to the calling environment. When an exception occurs inside a sub block, then it is handled by an exception handler within the sub block, terminating the sub block normally. Once the sub block gets terminated, the control is resumed in enclosing block immediately after the sub block END statement. But if there is no suitable exception handler found in the sub block for the raised exception, then the exception gets propagated to the successive enclosing blocks until the exception gets handled. If none of the enclosing blocks handle the exception, then finally the host environment will handle it but may not in a desired manner.

Advantages of propagating exceptions

a) Enclose specific exception handlers for the exceptions in their own block
b) More general exception handling can be done in the enclosing block



RAISE_APPLICATION_ERROR Procedure

This procedure is used to enable Oracle server to raise a pre-defined exception with a non-standard error code and error message. This avoids returning unhandled exceptions from the PL/SQL blocks. The syntax for using this procedure is:

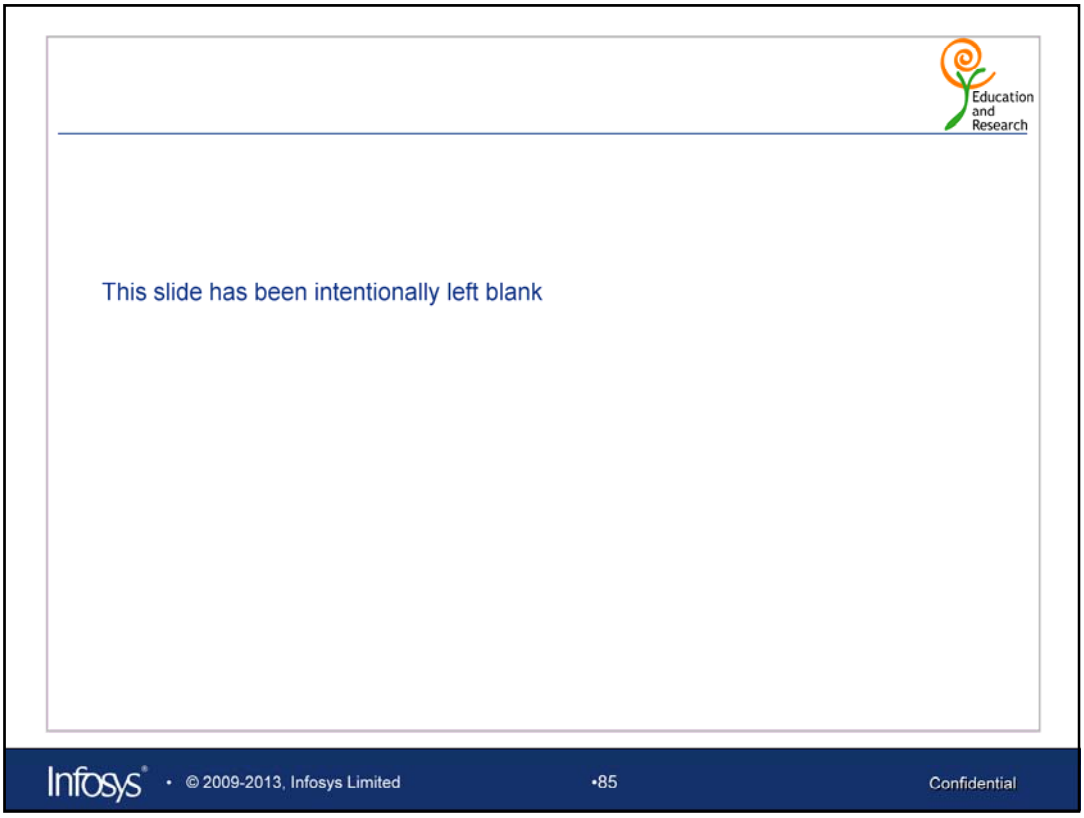
RAISE_APPLICATION_ERROR(<error_number>, <message>[, <{TRUE|FALSE}>);

error_number

- It is a number specified by the user. The range of numbers used for this is -20000 to -20999

message

- It is a used specified message.

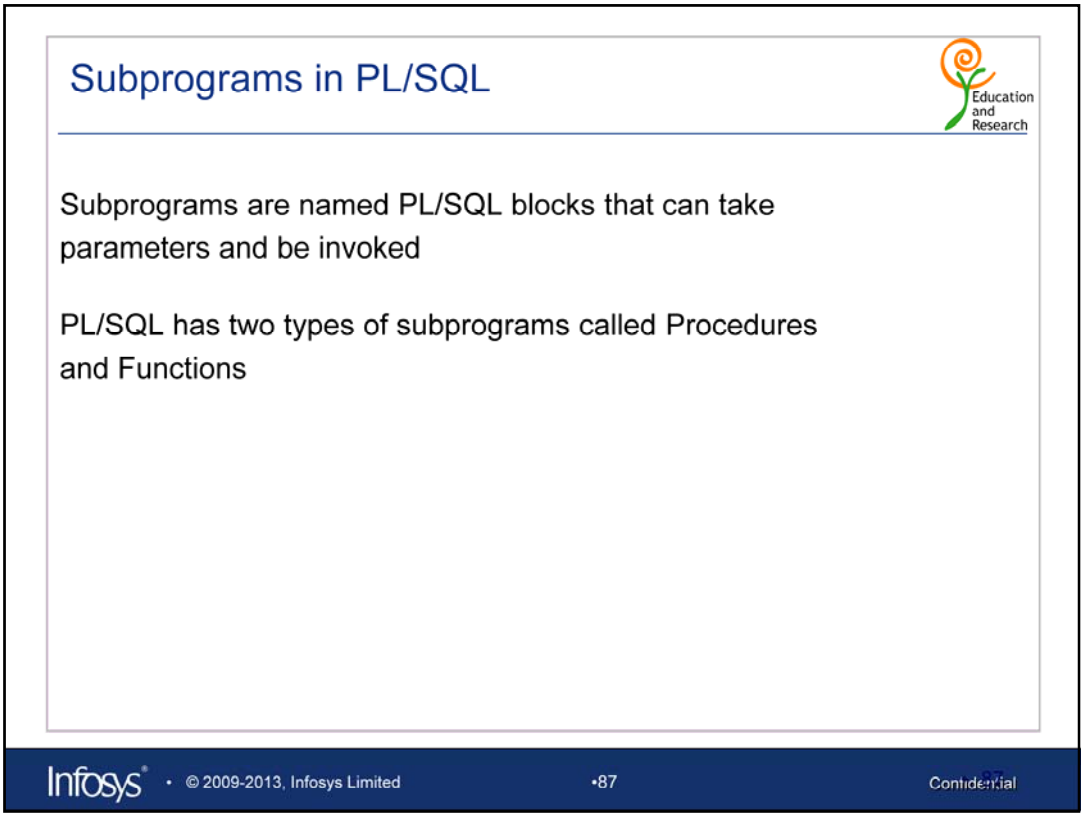


TRUE|FALSE

- It is optional
- If TRUE, error is placed in the stack of previous errors
- If FALSE, error is overwrites all previous errors

RAISE_APPLICATION_ERROR procedure can be used in both the executable section as well as the exception section of the PL/SQL block. The error returned using this procedure is consistent with the Oracle server error.

Subprograms in PL/SQL

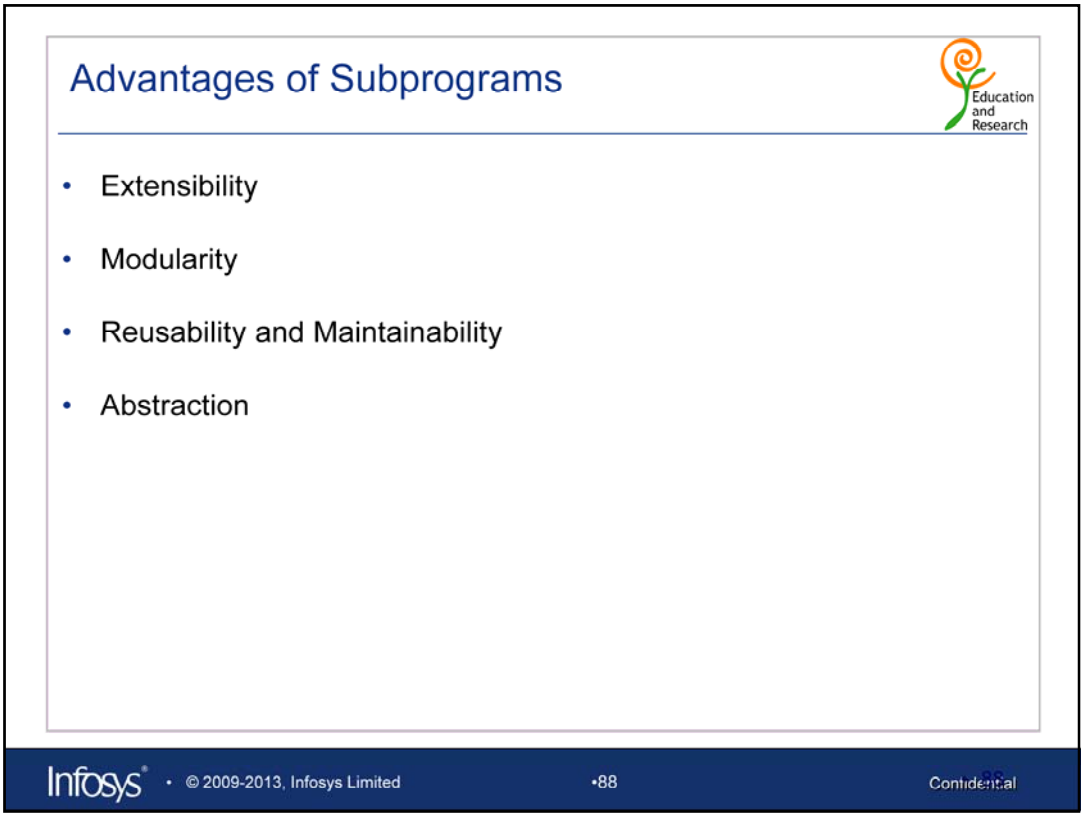


In PL/SQL, there are two types of subprograms called procedures and functions. Subprograms allow you to decompose a program into logical units that provide specific services or perform specific operations.

Procedures and Functions can be thought of as named anonymous blocks that accept values, return values and be called from other blocks, procedures or functions.

The basic difference between a procedure and a function is that a procedure is used to accept and return parameters and perform a action whereas a function is intended to accept parameters, compute a value and return the value to the caller.

Procedures and Functions have a declarative part, an executable part and an optional exception handling part.



How to use Procedures and Functions

Procedures and Functions can be created using any Oracle tool that supports PL/SQL.

Stored procedures and functions provide

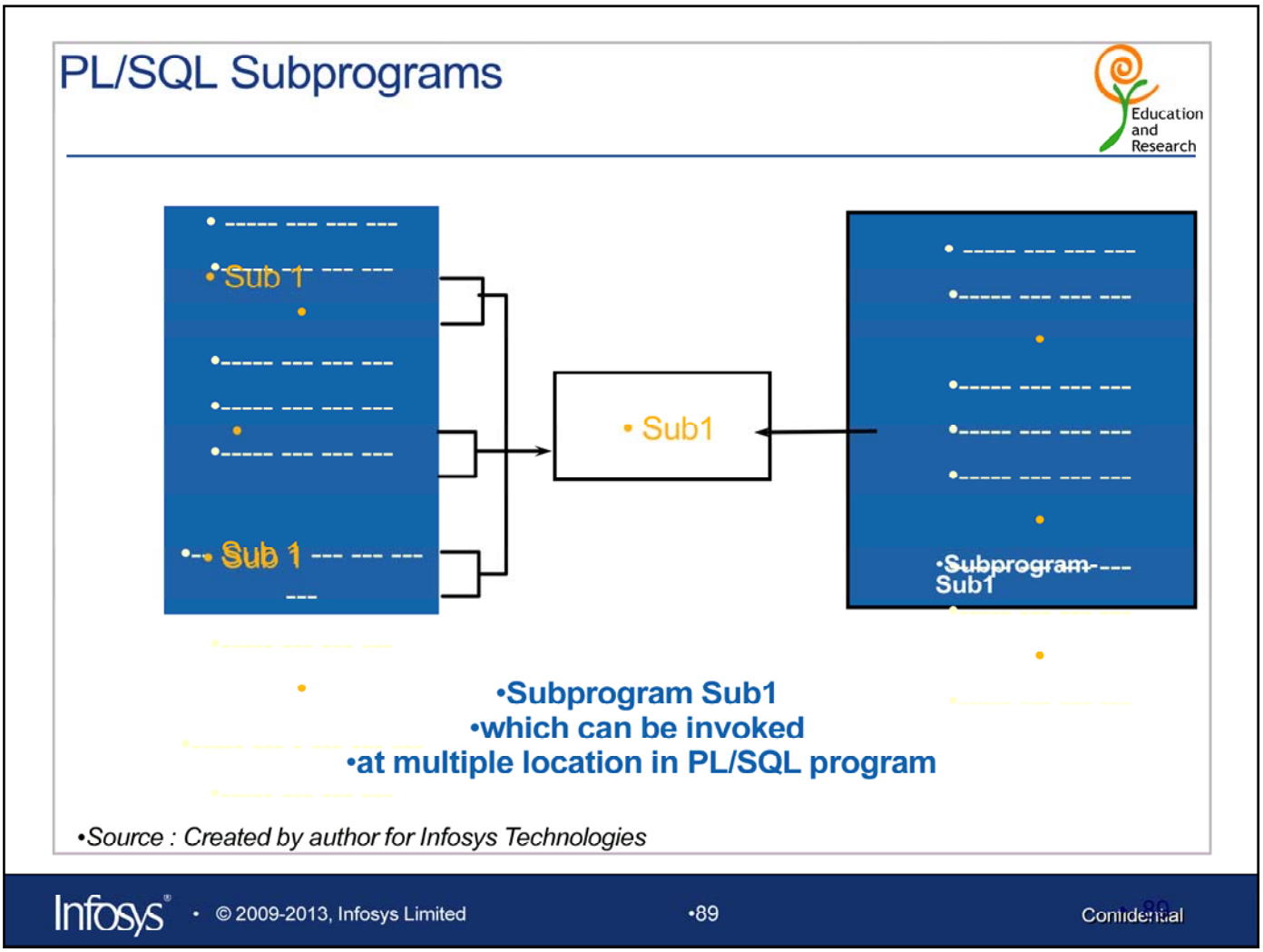
Higher Productivity

Better Performance

Memory savings

Application integrity

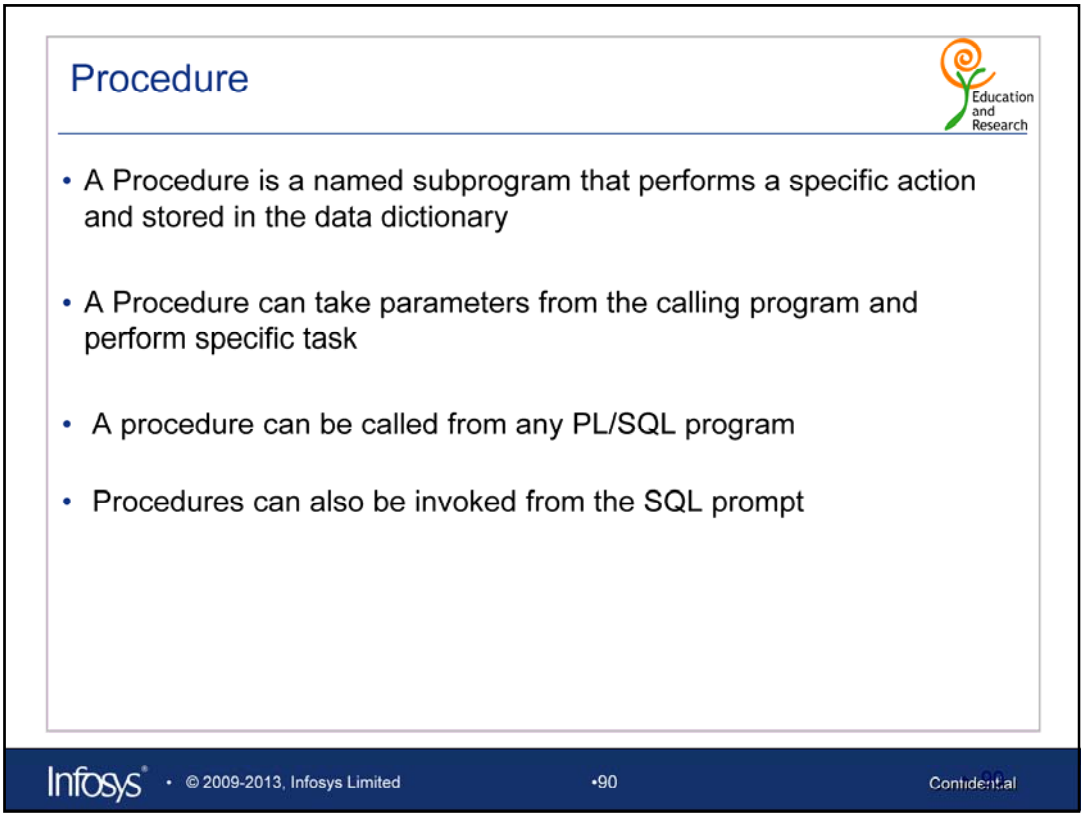
Tighter security



Subprograms

In the above slide it explains how you can reuse subprograms instead of writing same PL/SQL statement repeatedly in a PL/SQL block.

When you have repetitive PL/SQL statements in a Program, you can replace them by writing the repeated code in a sub-program and call the sub-program in place of the repeated code

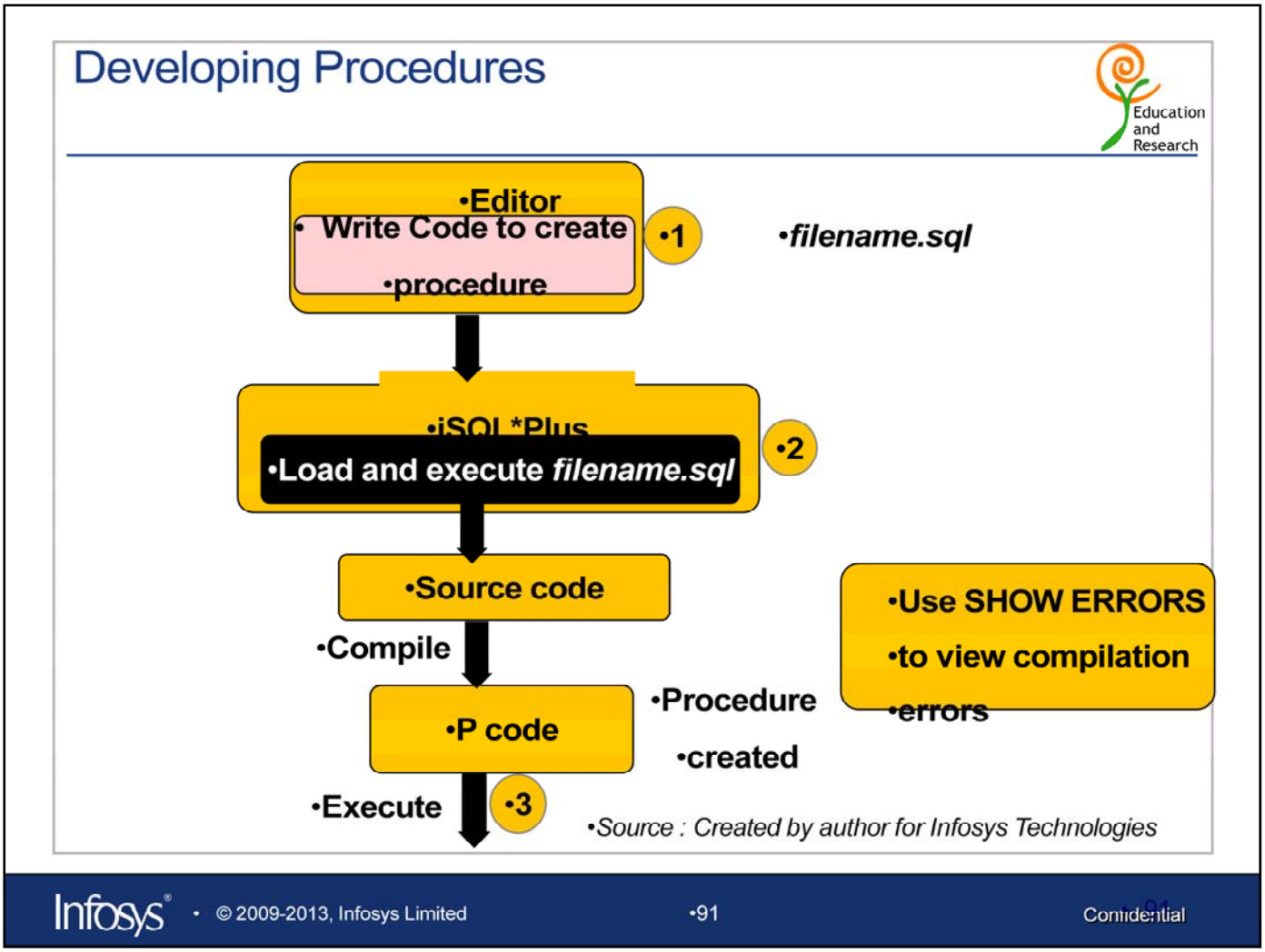


Definition of a Procedure

A procedure is a named PL/SQL block that can accept parameters or arguments and can be invoked from SQL prompt or PL/SQL block.

A procedure contains a header, a declaration section, an executable section, and an optional exception-handling section.

A procedure can be compiled and stored in the database as a schema object. The main advantages of creating a procedures is reusability and maintainability of code. We can create the procedure once and the same can be used in any number of applications any number of times. If the requirements change, only the procedure needs to be updated.



Developing Procedures

Step 1.Enter the code to create a procedure (CREATE PROCEDURE statement) in a system editor or word processor and save it as a SQL script file (.sql extension).

Step 2. Compile the code: Using SQL*Plus, load and run the SQL script file. The source code is compiled into P code and the procedure is created.

A script file with the CREATE PROCEDURE (or CREATE OR REPLACE PROCEDURE) statement enables you to change the statement if there are any compilation or run-time errors, or to make subsequent changes to the statement. You cannot successfully invoke a procedure that contains any compilation or run-time errors. In SQL*Plus, use SHOW ERRORS to see any compilation errors. Running the CREATE PROCEDURE statement stores the source code in the data dictionary even if the procedure contains compilation errors. Fix the errors in the code using the editor and recompile the code.

Step 3. Execute the procedure to perform the desired action. After the source code is compiled and the procedure is successfully created, the procedure can be executed any number of times using the EXECUTE command from SQL*Plus. The PL/SQL compiler generates the pseudocode or P code, based on the parsed code. The PL/SQL engine executes this when the procedure is invoked.

Syntax for Writing Procedure

```
CREATE [OR REPLACE] PROCEDURE <PROCEDURE_NAME>
(<arg1> [mode] <datatype>,...)
  IS | AS
  [local declaration];
          .
          .
          .
BEGIN
  PL/SQL executable statements;
[EXCEPTION
    exception handlers]
END [PROCEDURE_NAME];
```


Syntax to create a procedure is given in the PPT, where

Procedure name: is the name used to identify the procedure

Local declarations : Optionally declare your local variables, constants, cursors etc.

EXCEPTION : optional error handling section

Parameter : has the following syntax

<var_name>[IN| OUT | INOUT] data type [{:= | DEFAULT} value]

where, var_name is the unique parameter name. Each parameter is assigned an optional mode, IN forces the parameter to be considered for input only.

OUT forces a parameter to be considered for output only,

INOUT allows a parameter to be considered both for input and output.

Default mode is IN.

Data type is a valid PL/SQL data type.

The parameters can be assigned a DEFAULT value , thus you can have an option to omit the parameter during the call also.

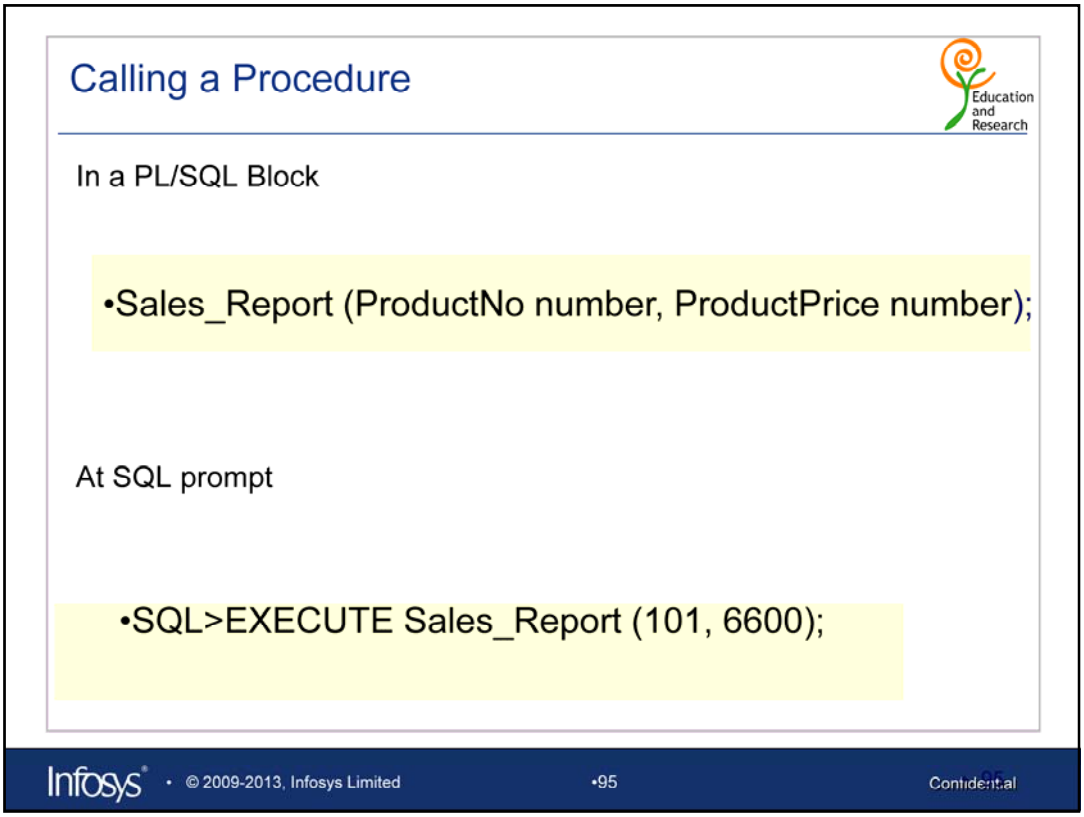
Example of a Procedure program

/* Procedure to update Prod_Price of a given Product Number*/

```
•CREATE or REPLACE PROCEDURE Sales_Report (Product_No number,
• Product_Price number) IS
•TempProd_Price  NUMBER;
•Product_Missing  EXCEPTION;
•BEGIN
• SELECT Prod_Price INTO TempProd_Price FROM SALES WHERE Prod_No =
• Product_No;
•IF TempProd_Price IS NULL THEN
•   RAISE Product_Missing;
•ELSE
•UPDATE SALES set Prod_Price = Prod_Price + Product_Price where Prod_No =
•Product_No;
•   END IF;
```

Example Contd..

```
EXCEPTION
    WHEN Product_Missing THEN
    dbms_output.put_line (Product_No || ' has Prod_Price as NULL');
    WHEN NO_DATA_FOUND THEN
    dbms_output.put_line (Product_No || ' No such Product');
END Sales_Report;
```



You can call a stored procedure from any of the following constructs:
Anonymous Block, Procedure or Function
Stored procedure or function
Database Trigger
Pre Compiler Application
OCI application
Oracle tool(oracle forms)

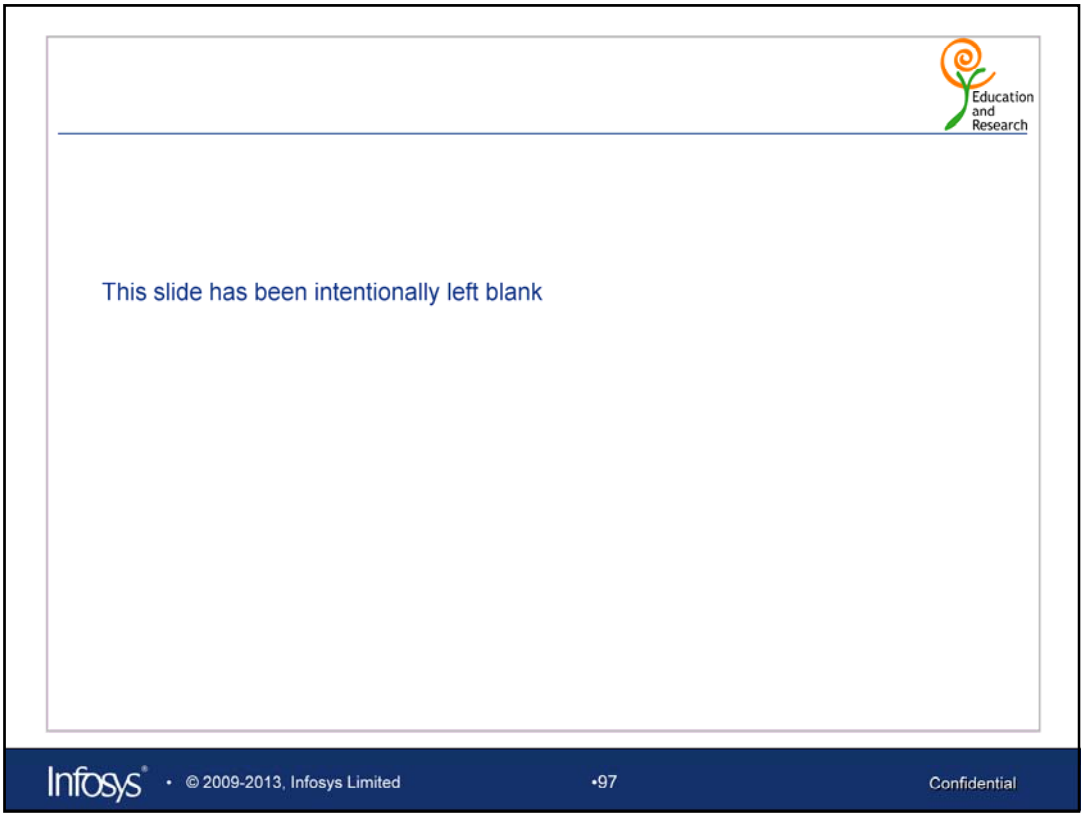
Stored Functions

- A Function is a named PL/SQL block that perform some specific task
- A Function can take parameters from the calling program and perform specific task and return value to the calling program
- A function can be stored in the database as a schema object for repeated execution.
- A function is called as part of an expression.

Stored Functions

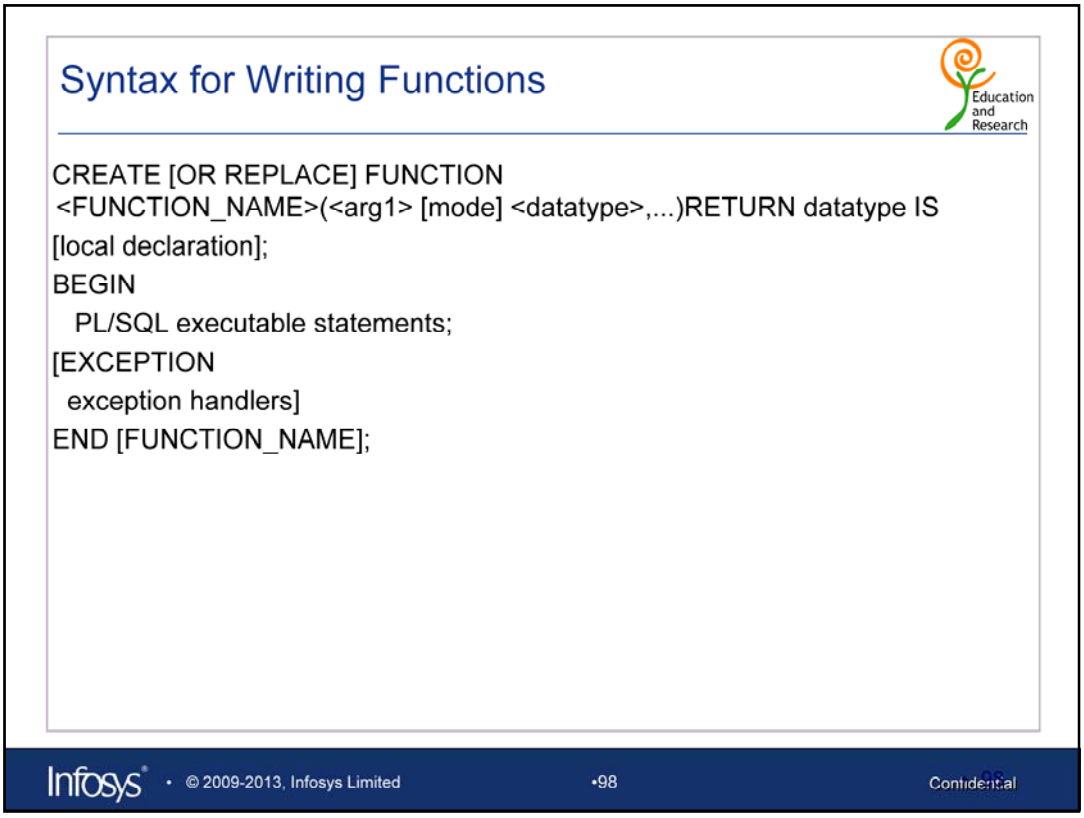
A function is a named PL/SQL block that can accept parameters and be invoked. Generally speaking, you use a function to compute a value. Functions and procedures are structured alike. A function must return a value to the calling environment, whereas a procedure returns zero or more values to its calling environment. Like a procedure, a function has a header, a declarative part, an executable part, and an optional exception-handling part. A function must have a RETURN clause in the header and at least one RETURN statement in the executable section.

Functions can be stored in the database as a schema object for repeated execution. A function stored in the database is referred to as a stored function. Functions can also be created at client side applications. This lesson discusses creating stored functions. Refer to appendix “Creating Program Units by Using Procedure Builder” for creating client-side applications.



Functions promote reusability and maintainability. When validated they can be used in any number of applications. If the processing requirements change, only the function needs to be updated.

Function is called as part of a SQL expression or as part of a PL/SQL expression. In a SQL expression, a function must obey specific rules to control side effects. In a PL/SQL expression, the function identifier acts like a variable whose value depends on the parameters passed to it.



A function is a subprogram like a procedure that is used to return one and only one value.

A function can be called from an anonymous block, procedure and other functions

The Syntax for creating a function is provided in the PPT, where

Function name : is the name used to identify the function

RETURN data type : used to declare the data type of the return value

Local Declarations : to optionally declare your local variables, constants, cursors etc.

EXCEPTION: optional error handling section

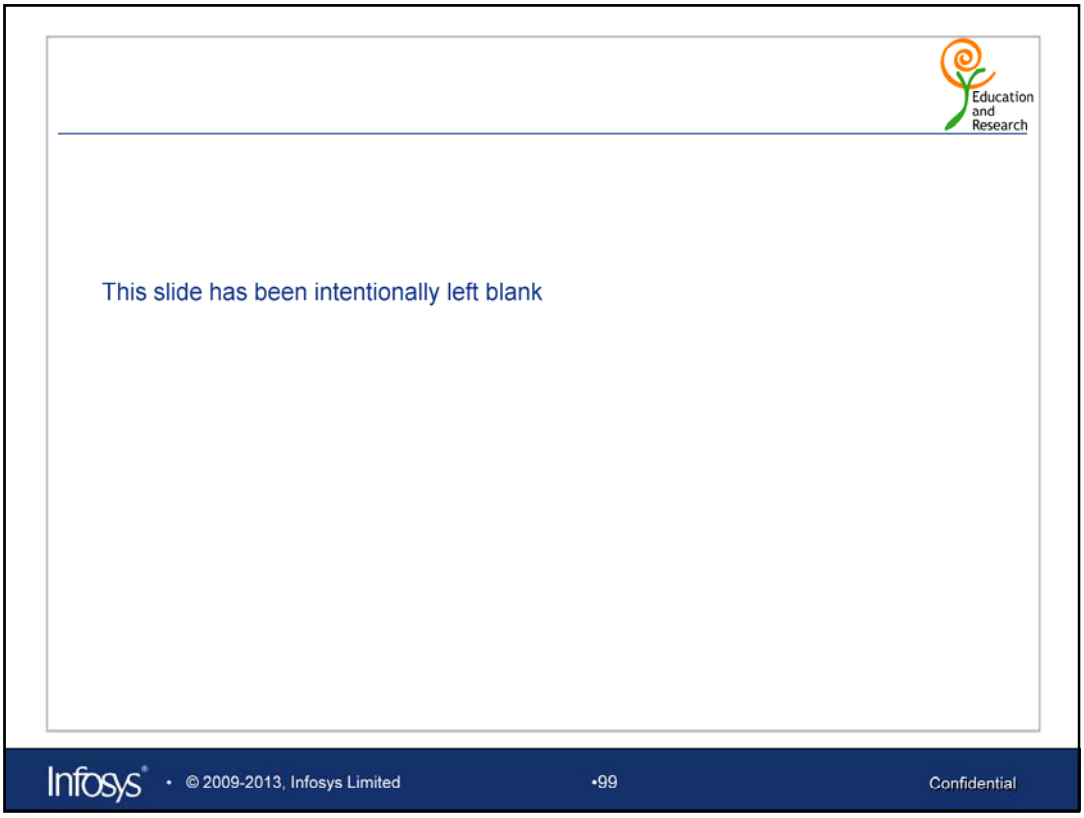
Parameter : has the following syntax

<var_name>[IN| OUT | INOUT] data type [{:= | DEFAULT} value]

where,

var_name is the unique parameter name. Each parameter is assigned an optional

mode,



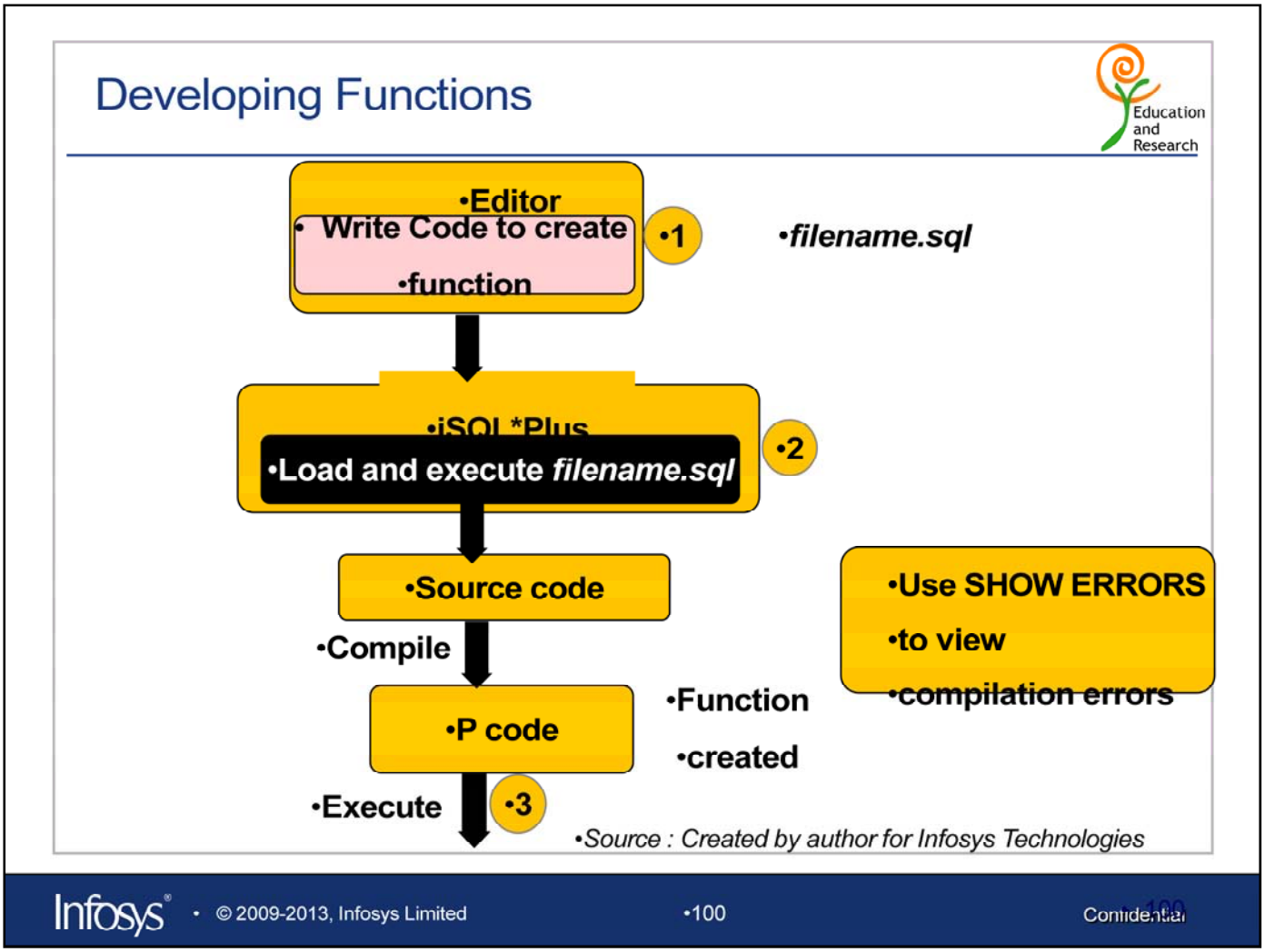
IN forces the parameter to be considered for input only.

OUT forces a parameter to be considered for output only and INOUT allows a parameter to be considered both for input and output

Default mode is IN.

Data type : is a valid PL/SQL data type.

The parameters can be assigned a DEFAULT value , thus you can have an option to omit the parameter during the call also.



Developing Functions

Step 1.Enter the code to create a function (CREATE FUNCTION statement) in a system editor or word processor and save it as a SQL script file (.sql extension).

Step 2 Compile the code: Using SQL*Plus, load and run the SQL script file. The source code is compiled into P code and the FUNCTION is created.

A script file with the CREATE FUNCTION (or CREATE OR REPLACE FUNCTION) statement enables you to change the statement if there are any compilation or run-time errors, or to make subsequent changes to the statement. You cannot successfully invoke a FUNCTION that contains any compilation or run-time errors. In SQL*Plus, use SHOW ERRORS to see any compilation errors. Running the CREATE FUNCTION statement stores the source code in the data dictionary even if the FUNCTION contains compilation errors. Fix the errors in the code using the editor and recompile the code.

Step 3.Execute the FUNCTION to perform the desired action. After the source code is compiled and the FUNCTION is successfully created, the FUNCTION can be executed any number of times using the EXECUTE command from SQL*Plus. The PL/SQL compiler generates the pseudocode or P code, based on the parsed code. The PL/SQL engine executes this when the FUNCTION is invoked.

Example of a Function program


```
•CREATE OR REPLACE FUNCTION SALES_Review (productno NUMBER)
•RETURN NUMBER IS

•incr_profit SALES.product_price%TYPE;
•net_profit SALES.product_price%TYPE;
•vproduct_no SALES.product_no%TYPE;
•vproduct_price SALES.product_price%TYPE;
•vproduct_comm SALES.product_comm%TYPE;

•BEGIN
```

Example Contd..

```
•select product_no,product_price,nvl(product_comm,0) into vproduct_no
•,vproduct_price,vproduct_comm from SALES where
•product_no = productno;
•net_profit := vproduct_price+vproduct_comm;
•IF vproduct_price <= 30000 then
•               incr_profit :=0.20 * net_profit;
•ELSIF vproduct_price > 30000 and vproduct_price <=60000 then
•               incr_profit:=0.30 * net_profit;
•ELSE
•               incr_profit := 0.40 * net_profit;
•END IF;
•return (incr_profit); END SALES_Review ;
```

Calling a Function

```
•DECLARE
•       incr_product_price number(7,2);
•BEGIN
•       incr_product_price:=SALES_Review (101001);
•       dbms_output.put_line (incr_product_price);
•END;
```

Advantages of User-Defined Functions in SQL Expressions

- Extend SQL where activities are too complex, too awkward, or unavailable with SQL
- Can increase efficiency when used in the `WHERE` clause to filter data, as opposed to filtering the data in the application
- Can manipulate character strings

Invoking User-Defined Functions from SQL Expressions

SQL expressions can reference PL/SQL user-defined functions. Anywhere a built-in SQL function can be placed, a user-defined function can be placed as well.

Advantages

Permits calculations that are too complex, awkward, or unavailable with SQL

Increases data independence by processing complex data analysis within the Oracle server, rather than by retrieving the data into an application

Increases efficiency of queries by performing functions in the query rather than in the application

Manipulates new types of data (for example, latitude and longitude) by encoding character strings and using functions to operate on the strings

Invoking Functions in SQL Expressions


```
•CREATE OR REPLACE FUNCTION calculate_price(amount IN NUMBER)
• RETURN NUMBER IS
•BEGIN
•   RETURN (amount+2000);
•END calculate_price;
•/

•SELECT product_id,product_name,product_amount,
•calculate_price(product_amount)  FROM   SALES
•                              WHERE  product_id = 1011;
```


Example

•The slide shows how to create a function tax that is invoked from a SELECT statement. The function accepts a NUMBER parameter and returns the tax after multiplying the parameter value with 0.08.

Locations to Call User-Defined Functions

- There are various location to call User Defined Functions
  - `SELECT` command
  - `WHERE` and `HAVING` clauses
  - `CONNECT BY`, `START WITH`, `ORDER BY`, and `GROUP BY` clauses
  - `INSERT` (Values) command
  - `UPDATE(Set)` command

Restrictions on Calling Functions from SQL Expressions

To be callable from SQL statements, a user-defined Stored function must follow the below points:-

- Function called from a SELECT statement or a parallelized INSERT, UPDATE, or DELETE statement, the function cannot modify any database tables.
- Function called from an INSERT, UPDATE, or DELETE statement, the function cannot query or modify any database tables modified by that statement.

To be callable from SQL statements, a user-defined
Stored function must follow the below points:-

Function called from a SELECT statement or a parallelized INSERT, UPDATE, or DELETE statement, the function cannot modify any database tables.

Function called from an INSERT, UPDATE, or DELETE statement, the function cannot query or modify any database tables modified by that statement.

Note: Only stored functions are callable from SQL statements. Stored procedures cannot be called.

Note: Functions that are callable from SQL expressions cannot contain OUT and IN OUT parameters. Other functions can contain parameters with these modes, but it is not recommended.

Restrictions on Calling Functions from SQL Expressions(Contd…)

- Function called from a SELECT, INSERT, UPDATE, or DELETE statement, the function cannot execute SQL transaction control statements (such as COMMIT), session control statements (such as SET ROLE), or system control statements (such as ALTER SYSTEM).
- It also cannot execute DDL statements (such as CREATE) because they are followed by an automatic commit.

Function called from a SELECT, INSERT, UPDATE, or DELETE statement, the function cannot execute SQL transaction control statements (such as COMMIT), session control statements (such as SET ROLE), or system control statements (such as ALTER SYSTEM).

Also it cannot execute DDL statements (such as CREATE) because they are followed by an automatic commit.

Restrictions on Calling Functions from SQL:- Example


```
•CREATE OR REPLACE FUNCTION product_sales(amount NUMBER)
•  RETURN NUMBER IS
•BEGIN
•   INSERT INTO SALES(product_id,product_name,product_price)
•              VALUES(101, 'BTM', '20000');
•   RETURN (amount+ 1100);
•END;
```

```
•UPDATE SALES SET product_price= product_sales(2000)
• WHERE product_id = 102;
```


Restrictions on Calling Functions from SQL: Example

The function contains one insert statement to inserts a new record into the SALES table.

Also the function is invoked from the UPDATE statement that updates the product price of product id of 102.

The UPDATE statement returns an error saying that the table is mutating.

Argument Modes

IN

The IN parameter allows the user to pass values to the called subprogram. Inside the subprogram, the IN parameter acts like a constant, thus cannot be modified (Default behavior)

OUT

The OUT parameter lets the user return values to the calling block.

It specifies that the called procedure passes a value for this argument back to its calling environment after execution.

IN OUT

The IN OUT parameter lets the user pass initial values to the called subprogram and returns updated values to the calling block after execution.

Comparison of Parameter Modes

•IN	•OUT	•IN OUT
•Default mode	•Must be specified	•Must be specified
•Passes values to a subprogram	•Returns values to the caller	•Passes initial values to a subprogram and returns updated values to the caller
•Formal parameter acts like a constant	•Formal parameter acts like a variable •	•Formal parameter acts like an initialized variable
•Actual parameter can be a constant, initialized variable, literal, or expression	•Actual parameter must be a variable	•Actual parameter must be a variable
•default value can be assigned	•default value •Cannot be assigned	•default value cannot be assigned

Creating Procedures with Parameters

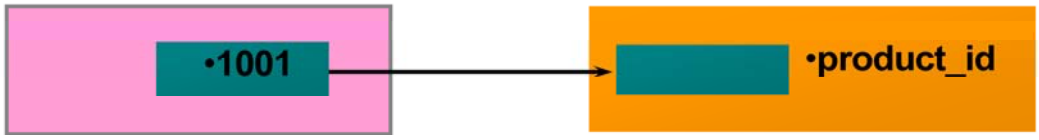
When you create the procedure, the formal parameter defines the value used in the executable section of the PL/SQL block, whereas the actual parameter is referenced when invoking the procedure. The parameter mode IN is the default parameter mode. That is, if mode is not specified for a parameter, the parameter is considered to be an IN parameter. The parameter modes OUT and IN OUT must be explicitly specified for parameters requiring them.

A formal parameter of IN mode cannot be assigned a value. That is, an IN parameter cannot be modified in the body of the procedure.

An OUT or IN OUT parameter must be assigned a value before returning to the calling environment. IN parameters can be assigned a default value in the parameter list. OUT and IN OUT parameters cannot be assigned default values.

By default, the IN parameter is passed by reference and the OUT and IN OUT parameters are passed by value. To improve performance with OUT and IN OUT parameters, the compiler hint NOCOPY can be used to request to pass by reference.

IN Parameters: Example




```
•CREATE OR REPLACE PROCEDURE increase_price
•  (product_id IN SALES.prodID%TYPE)
•IS
•BEGIN

•  UPDATE SALES SET  prodPRICE = prodPRICE*0.02
•                                  WHERE  prodID = product_id ;
•END increase_price;
•/SQL>EXECUTE increase_price(1001);
```

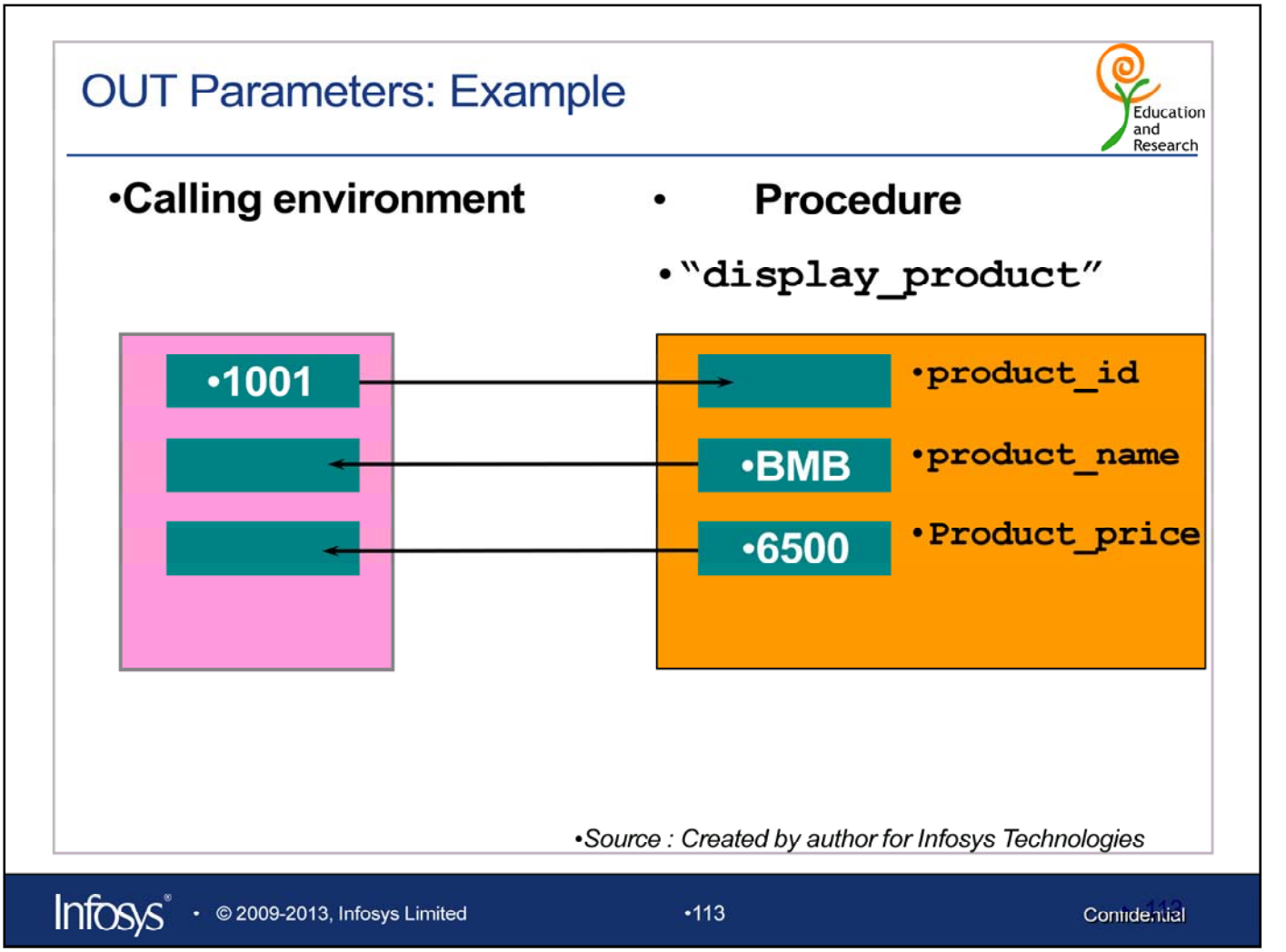

IN Parameters: Example

The example in the slide shows a procedure with one IN parameter.

Executing this procedure in iSQL*Plus creates the increase_price procedure. While invoking increase_price, it accepts one parameter for the PRODUCT ID and updates the SALES table's PRODUCT PRICE with a increase of 2 percent. To invoke a procedure in SQL*Plus, we use the EXECUTE command.

SQL>EXECUTE increase_price(1001);

IN parameters are passed as constants from the calling environment into the procedure. Attempts to change the value of an IN parameter result in an error.



OUT Parameters: Example

The example in the slide shows,

A procedure with one IN and two OUT parameters to retrieve information about product.

The procedure accepts a value for product id 1001 and retrieves the prodcuct name, product price from the SALES table into the two output parameters and return values to the calling environment. The code to create the display_product procedure is shown in the next slide.

OUT Parameters: Example


```
•CREATE OR REPLACE PROCEDURE display_product
• (product_id        IN   SALES.prod_id%TYPE,
•  product_name    OUT  SALES.prod_name%TYPE,
•  product_price    OUT  SALES.prod_price%TYPE )IS
•BEGIN
• SELECT   prodname, prodname
•  INTO    product_name, product_price
•  FROM    SALES
•  WHERE   prod_id = product_id;
•END display_product;
```

Viewing OUT Parameters from SQL*PLUS

- Declare host variables, and execute the hike_price procedure, and display the value of the global prod_name variable.

```
•VARIABLE prod_name   VARCHAR2(25)
•VARIABLE prod_price  NUMBER
•VARIABLE prod_comm   NUMBER
•EXECUTE hike_price(1001, :prod_name, :prod_price,:prod_comm)
•PRINT prod_name
```


To view the Value of OUT Parameters from SQL*Plus

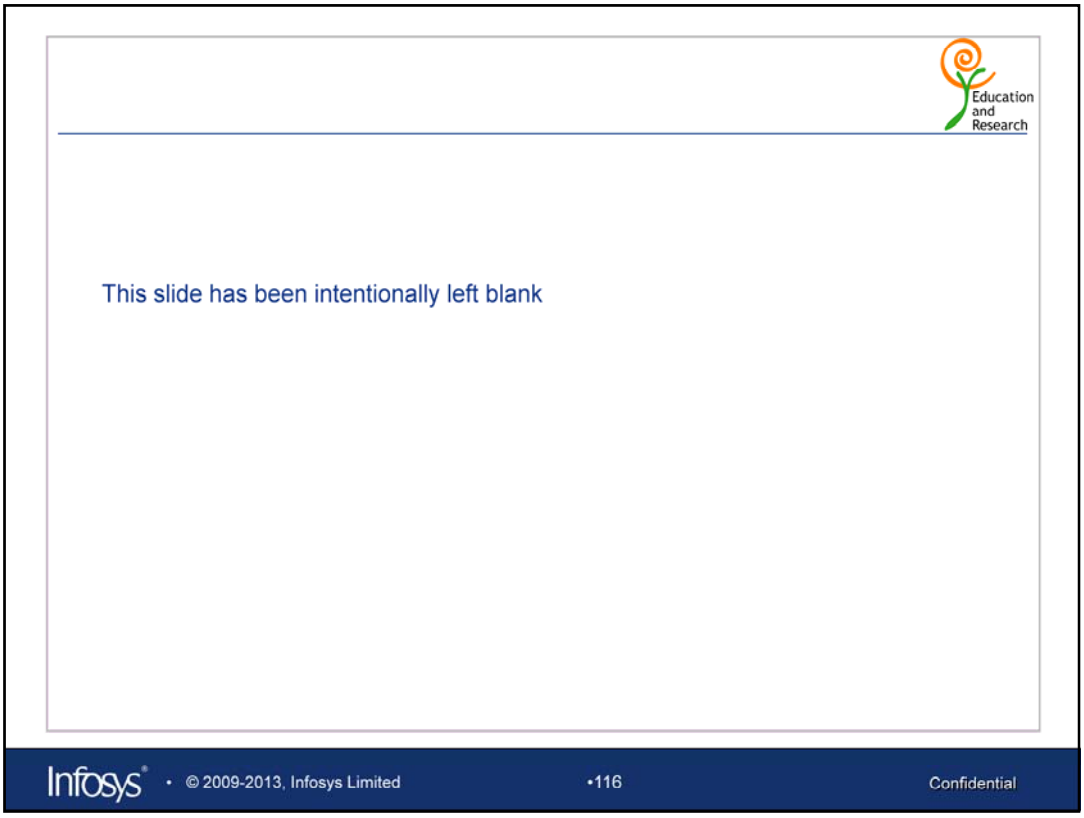
1. Run the script file to compile generate the source code.
2. use the VARIABLE command to create host variables in SQL*Plus.
3. Invoke the hike_price procedure, and supply these host variables as the OUT parameters.
4. To view the values passed from the procedure to the calling environment, use the PRINT command.

Example in the slide shows the value of the prod_name variable passed back to the calling environment. The other variables can be viewed, either individually, or with a single PRINT command.

PRINT prod_name prod_price

Do not specify a size for a host variable of data type NUMBER when using the VARIABLE command. A host variable of data type CHAR or VARCHAR2 defaults to a length of one, unless a value is supplied in parentheses.

PRINT and VARIABLE are iSQL*Plus commands.



Note:

-use colon (:) to reference the host variables in the EXECUTE command.

-Passing a constant or expression as an actual parameter to the OUT variable causes compilation errors. For example:

EXECUTE hike_price(1001, :prod_name, hike+1000, :g_price)

causes a compilation error.

Viewing IN OUT Parameters


```
•Create or replace procedure modify_PROD_Price(prod_price IN OUT
• NUMBER)
•IS
•BEGIN
• prod_price:= 75000;
• END;
```

```
DECLARE
•product_price number(10):= 50000; BEGIN
•DBMS_OUTPUT.PUT_LINE(product_price);
• PHONE_PROC(product_price);
•DBMS_OUTPUT.PUT_LINE(product_price);END;
```


Create the procedure modify_PROD_Price and call the procedure from an anonymous block.

Methods for Passing Parameters

- While calling a subprogram , you can write the actual
- parameters using positional, named or combination
- notation
  - Positional: The order of actual parameters would be same as order of formal parameters.
  - Named: The order of actual parameters would be in arbitrary order by associating each with its corresponding formal parameter.
  - Combination: The order of the actual parameters as positional and some as named.

Parameter Passing Methods

For a procedure that contains multiple parameters, you can use a number of methods to specify the values of the parameters.

Method	Description
Positional	Lists values in the order in which the parameters are declared
Named association	Lists values in arbitrary order by associating each one with its parameter name, using special syntax (=>)
Combination	Lists the first values positionally, and the remainder using the special syntax of the named method

Methods for Passing Parameters:- Example


```
PROCEDURE Sales_Report (product_no INTEGER, product_amount
  REAL) IS
```

You can call the procedure Sales_Report in four logically equivalent ways

```
Sales_Report product, amt); -- positional notation
Sales_Report(product_amount => amt, product_no => prod_no); --
  named notation
Sales_Report(product_no => prod_no, product_amount => amt); --
  named notation
Sales_Report product, product_amount => amt); -- mixed notation
```


In mixed notation the first parameter uses positional notation, and the second parameter uses named notation. Positional notation must precede named notation. The reverse is not allowed. For example, the following procedure call is illegal:

Sales_Report (product_no => product, amt); -- illegal

DEFAULT Option for Parameters

```
•CREATE OR REPLACE PROCEDURE add_product
• (product_name  IN SALES.prod_name%TYPE  DEFAULT 'abc',
•  product_no   IN SALES.prod_no%TYPE DEFAULT 1000)
•IS
•BEGIN
• INSERT INTO SALES(prod_id,prod_name, prod_no)
• VALUES (prod_seq.NEXTVAL, product_name,product_no);
•END add_product;
•/
```


Example of Default Values for Parameters

You can initialize IN parameters to default values. That way, you can pass different numbers of actual parameters to a subprogram, accepting or overriding the default values as you please. Moreover, you can add new formal parameters without having to change every call to the subprogram.

Execute the statement in the slide to create the ADD_PRODUCT procedure. Note the use of the DEFAULT clause in the declaration of the formal parameter. You can assign default values only to parameters of the IN mode. OUT and IN OUT parameters are not permitted to have default values.

If an actual parameter is not passed, the default value of its corresponding formal parameter is used. Consider the calls to the above procedure that are depicted in the next page.

Examples of Passing Parameters


```
•BEGIN
• add_product;
• add_product ('BMB', 1500);
• add_product ( product_no => 1400, product_name =>'ABC');
• add_product ( product_no => 1200) ;
•END;
•/
•SELECT prod_id, prod_name,prod_no
•FROM SALES;
```

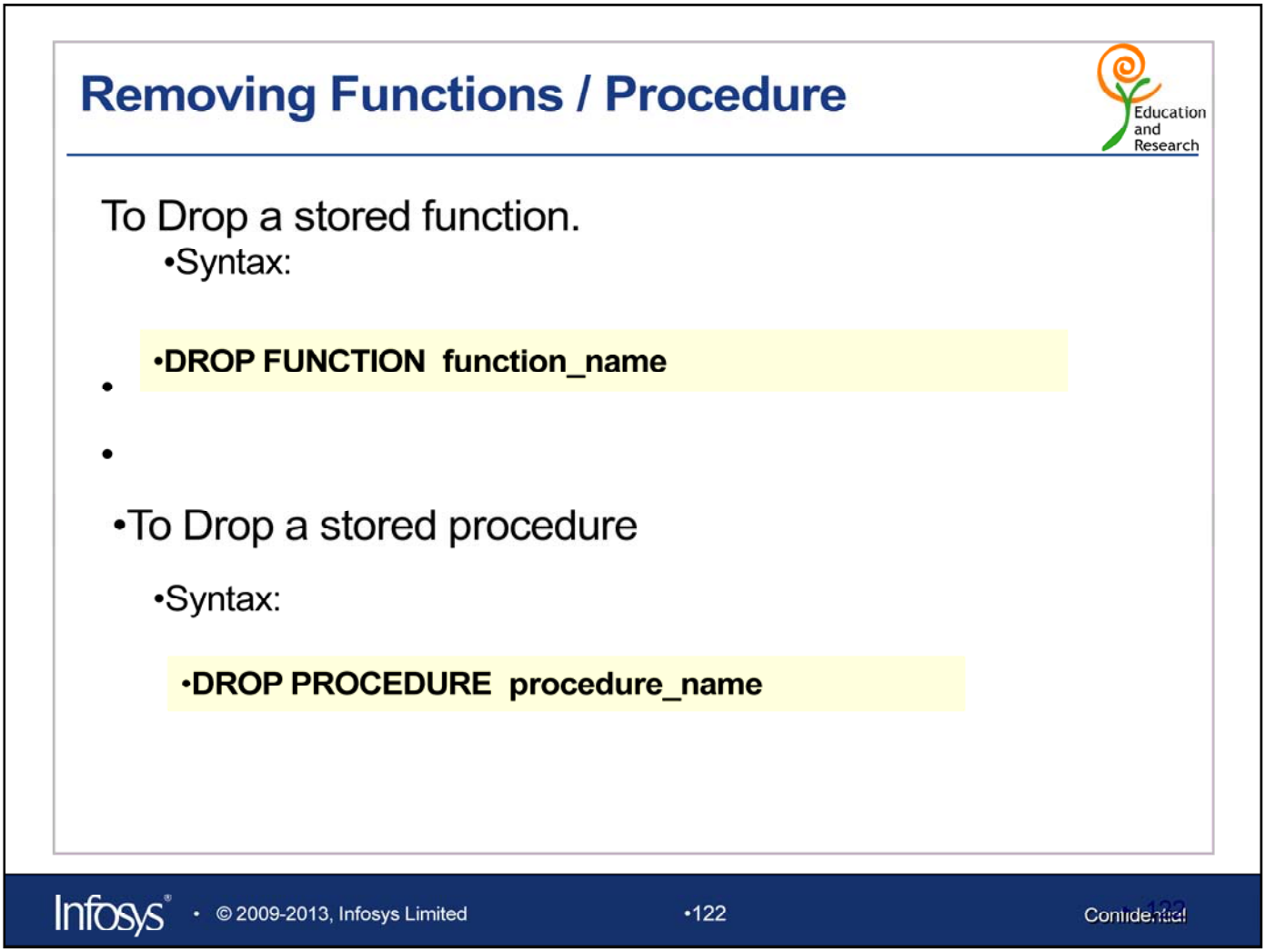

Example of Default Values for Parameters (continued)

The anonymous block above shows the different ways the ADD_PRODUCT procedure can be invoked, and the output of each way the procedure is invoked.

Usually, you can use positional notation to override the default values of formal parameters. However, you cannot skip a formal parameter by leaving out its actual parameter.

Note: All the positional parameters should precede the named parameters in a subprogram call. Otherwise, you will receive an error message, as shown in the following example:

EXECUTE add_product(product_name=>'new product', 'new productno')



Removing Functions

When a stored function/procedure are no longer required, you can drop it a SQL statement in SQL*Plus to drop it. To remove a stored function, Use the SQL command

DROP FUNCTION FUNCTION_NAME

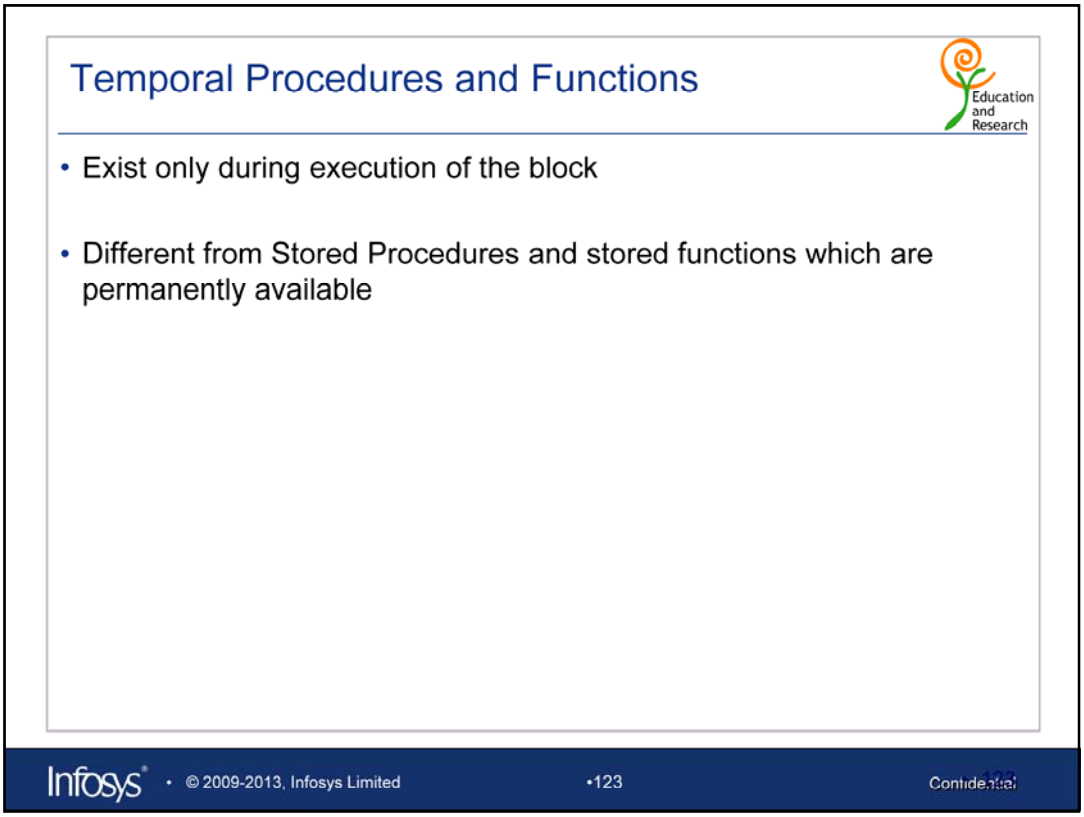
DROP PROCEDURE PROCEDURE_NAME

CREATE OR REPLACE Versus DROP and CREATE

The REPLACE clause in the CREATE OR REPLACE syntax is equivalent to dropping a function and re-creating it. When you use the CREATE OR REPLACE syntax, the privileges granted on this object to other users remain the same. When you DROP a function and then create it again, all the privileges granted on this function are automatically revoked.

Removing Procedures

When a stored procedure is no longer required, you can use a SQL statement to drop it. To remove a server-side procedure by using SQL*Plus, execute the SQL command DROP PROCEDURE.Issuing rollback does not have an effect after executing a data definition language (DDL) command such as DROP PROCEDURE, which commits any pending transactions.



Procedures and Functions can be created using any Oracle tool that supports PL/SQL.

Procedures and Functions can be declared as part of any anonymous block, procedure, function or package.

A procedure or function must be declared at the end of a declarative section after all other objects are declared.

Procedures and Functions that are declared locally in other blocks, procedures or functions can be accessible only from that program.

However Procedures and Functions can be made globally accessible once they are recorded as part of the database.

Advantages

Procedures and Functions like in any modular programming language provide modularity, extensibility, reusability, maintainability and abstraction.

Example


```
DECLARE
  studentNo STUDENT.Stud_No%TYPE;
  StudGrade CHAR;
  StudMarks STUDENT.Stud_Marks%TYPE;

  PROCEDURE Student_Grade( StudNo IN STUDENT.Stud_No%TYPE :=0
  ,Grade OUT  CHAR) IS
  BEGIN
          SELECT Stud_Marks into StudMarks FROM STUDENT WHERE
                    Stud_No = StudNo;
           IF StudMarks >=85 and StudMarks <=75 THEN
                    Grade:='A';
  ELSE
                    Grade:='B';
  END IF;
  EXCEPTION
  WHEN NO_DATA_FOUND THEN
                    Grade := 'C';
  END;
```

Example (Contd..)

```
BEGIN
  studentNo :=&STUDENTNO;
  Student_Grade(studentNo,StudGrade);
  IF StudGrade = 'C' THEN
        dbms_output.put_line('Student Not Found');
  ELSE
        dbms_output.put_line('Student Grade is '||StudGrade);
  END IF;
END;
```

Data Dictionary View

Education and Research

- To display all Procedures and Functions

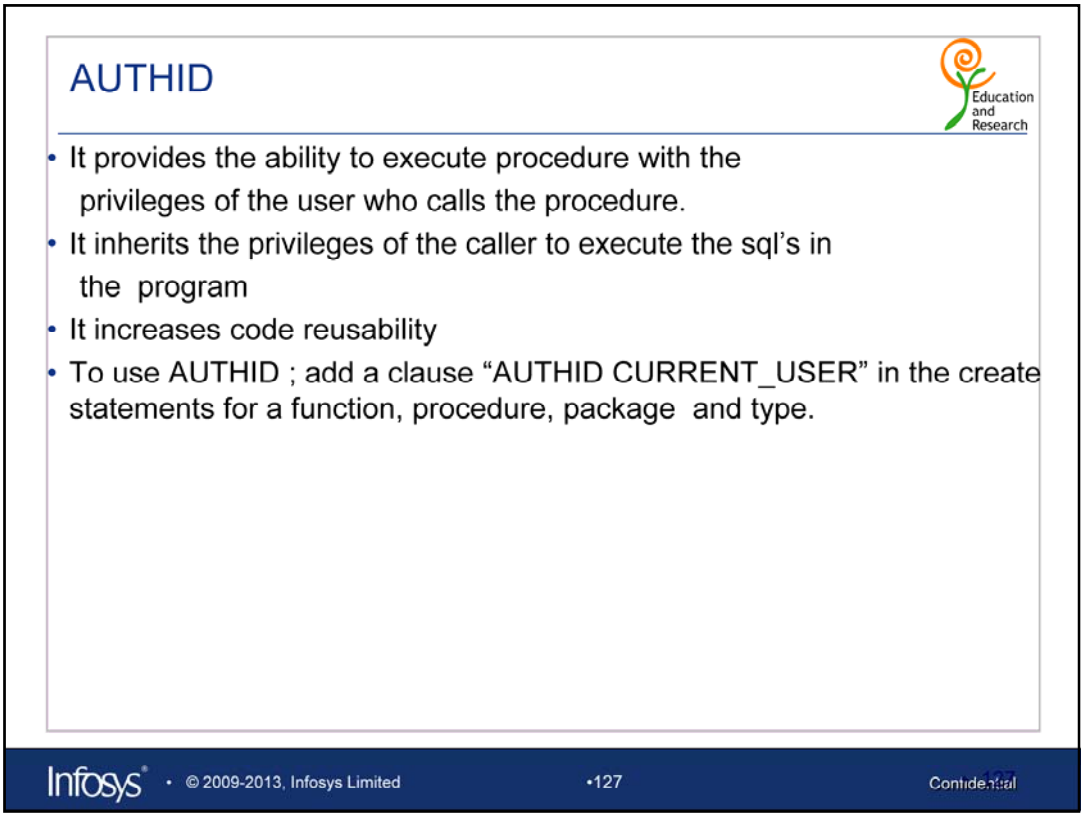
```
•SELECT  *
•FROM  user_objects
•WHERE object_type in ('PROCEDURE','FUNCTION')
•ORDER BY object_name;
```

- To display the source code of Procedures andFunctions

```
•SELECT  TYPE, TEXT, NAME
•FROM  user_source
•WHERE  name = 'ABC'
```


Example

The example in the slide displays the names of all the procedures and functions that you have created.

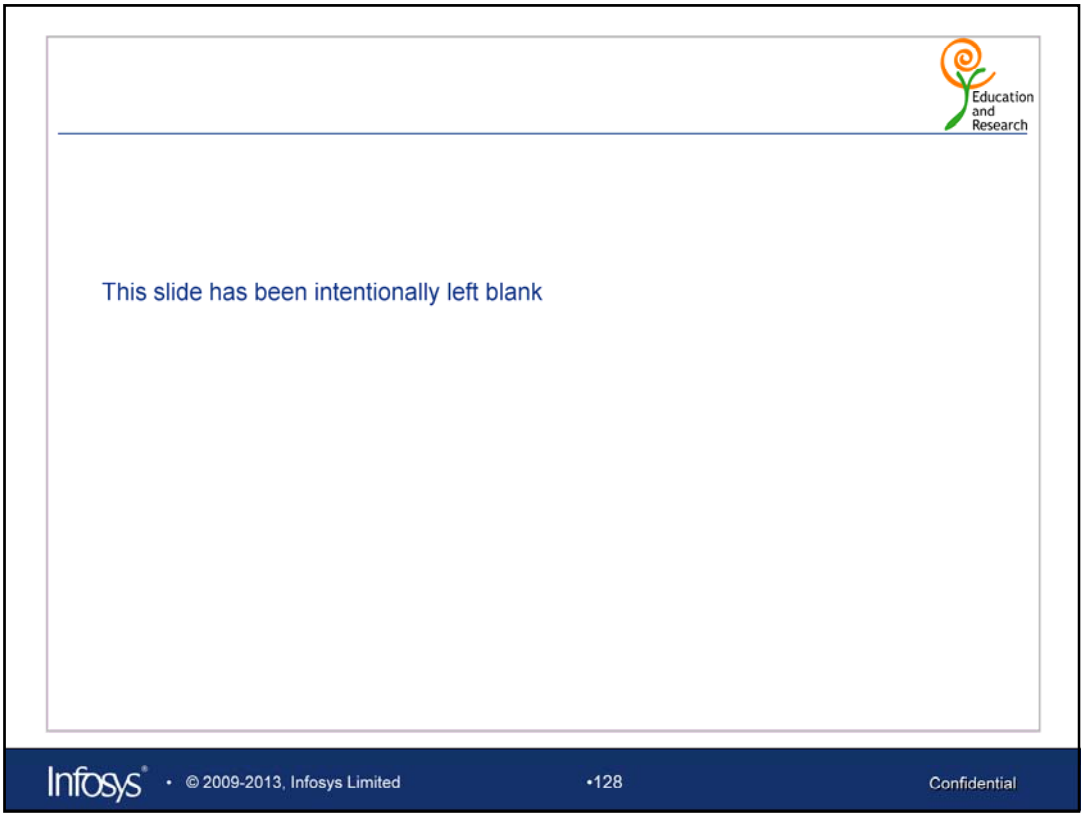


By default a stored procedure or method executes with the privileges of its owner. In Oracle 9i it is possible to execute the programs with privileges of the caller. An Invoker's rights program inherits the privileges and name resolution context of the user calling the program. It enhances the code reuse and centralizes the data administration.

The programs can be made to inherit Invoker's rights by adding the AUTHID clause in the create statements of a function, procedure, package and type. We have to specify AUTHID as CURRENT_USER.

Ex:

```
CREATE OR REPLACE PROCEDURE delemp (tempno  in
VARCHAR2)
AUTHID CURRENT_USER
IS
BEGIN
DELETE FROM employee WHERE empno=tempno
END;
```

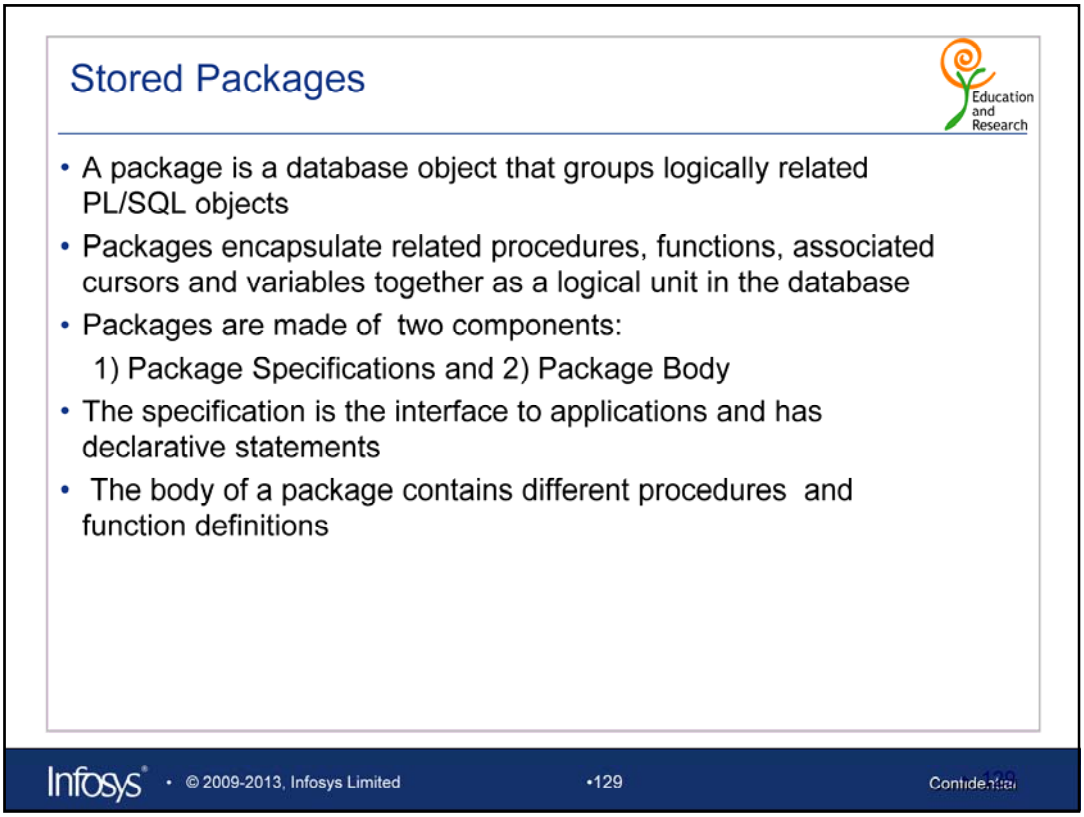


In the example above, oracle will refer the employee table in the user's schema, who invokes this procedure. If AUTHID clause is not specified, Employee table from the definer's schema is referred.

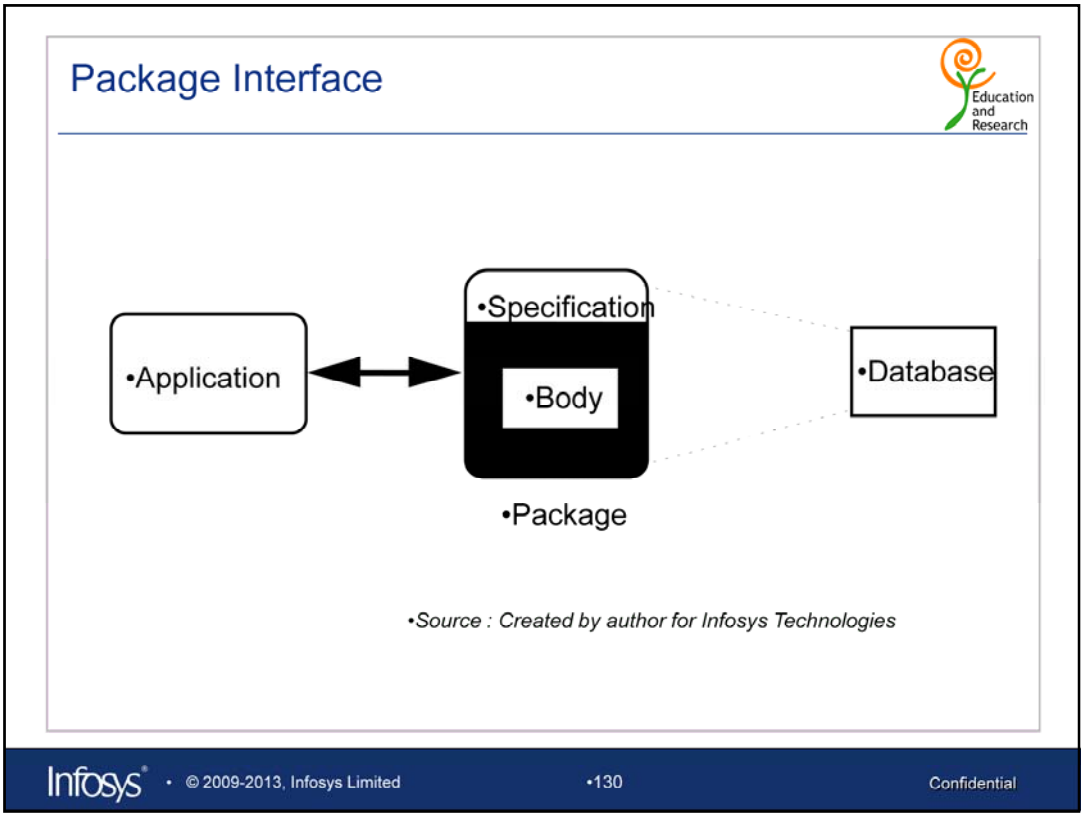
Name resolution happens in the following way

Names used in queries, DML statements, Dynamic SQL statements are resolved in the users schema.

All other statements such as calls to packages, functions and procedures are resolved in the definer's schema.



A package is like a source module in C language. A package may collect a set of related procedures and functions that serve as a subsystem to enforce specific business rules. A package typically consists of set of standardized data types, exceptions, variables, cursors, procedures or functions. Once created, a package acts as a library, where its contents can be shared among different applications.



A package is made up two parts, package specification and package body.

Constructs which are public are declared in the package specification and defined in the package body. Constructs which are private declared and defined exclusively within the package body.

Developing a Package

- To create the package specification, write the code within CREATE PACKAGE and run the SQL script file. The source code is compiled into P code and is stored within the data dictionary.
- To create the package body write the code within the CREATE PACKAGE BODY and run the SQL script file. The source code is compiled into P code and is stored within the data dictionary.
- Write and PL/SQL block to invoke public construct
- A package specification can exist without a package body, but a package body cannot exist without a package specification.

Developing a Package

There are three steps to developing a package,

To create the package specification, write the code within CREATE PACKAGE and run the SQL script file. The source code is compiled into P code and is stored within the data dictionary.

To create the package body write the code within the CREATE PACKAGE BODY and run the SQL script file. The source code is compiled into P code and is stored within the data dictionary.

Write and PL/SQL block to invoke public construct

A package specification can exist without a package body, but a package body cannot exist without a package specification.

Creating the Package Specification

•**Syntax:**

•**CREATE [OR REPLACE] PACKAGE** ***package_name***
•**IS|AS**
- ***public type and item declarations***
- ***subprogram specifications***

•**END** ***package_name;***

•**CREATE [OR REPLACE] PACKAGE BODY** ***package_name***
•**IS|AS**
- ***public type and item definition***
- ***private type and item declaration and definition***
- ***subprogram specifications***

•**END** ***package_name;***

- The REPLACE deletes and recreates the package specification and body
- by default variables are initialized by NULL
- All the items declared in a package specification are accessible to users who are granted privileges on the package.

How to Create a Package Specification and Package Body

To create packages, you declare all public constructs within the package specification and define within package Body, You can also declare and define private constructs within package Body

Specify the REPLACE option when the package specification already exists.

Initialize a variable with a constant value otherwise, the variable is initialized implicitly to NULL.

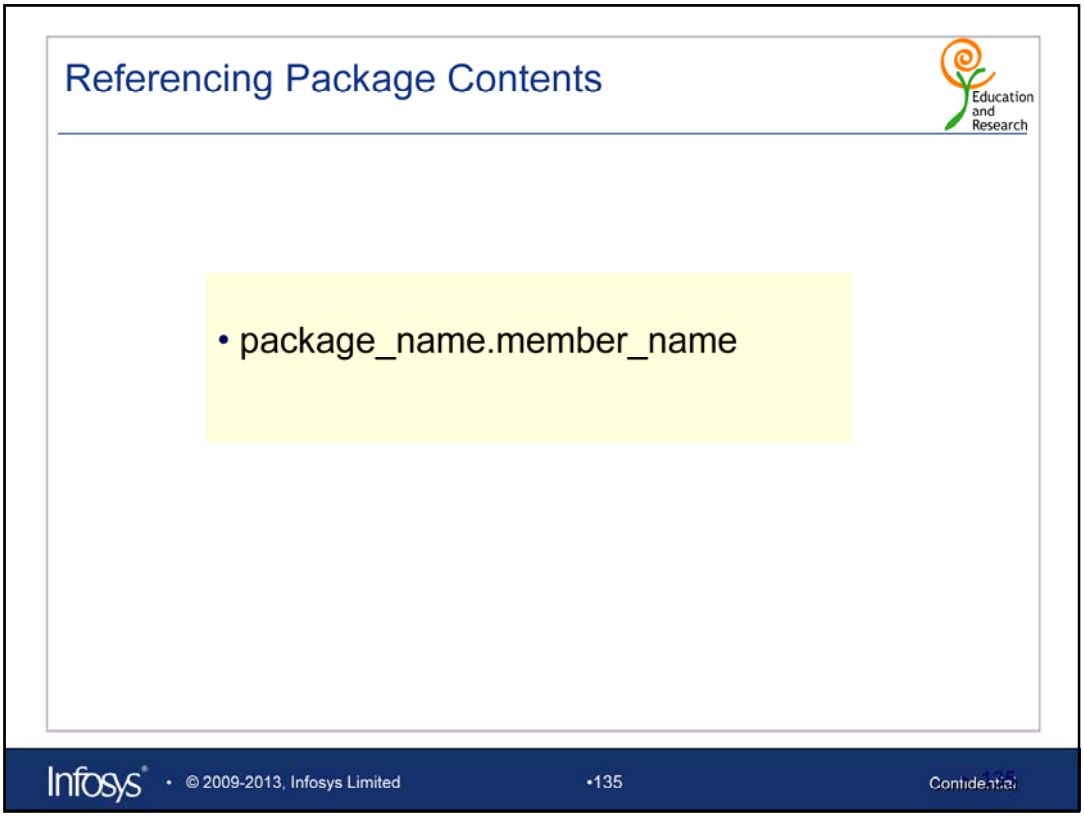
Parameter	Description
package_name	Name the package
public type and item declarations	Declare variables, constants, cursors, exceptions, or types
subprogram specifications	Declare the PL/SQL subprograms

Example

```
CREATE OR REPLACE PACKAGE PRODUCT_PACK AS

 PROCEDURE PRODUCT_SALES (PRODUCTCODE IN
  SALES.PRODUCTNO%TYPE);

 FUNCTION PRODUCT_PRICE( PROD_id NUMBER,PROD_amt NUMBER)
  RETURN   NUMBER;

END PRODUCT_PACK;
```

Example(Contd..)

```
CREATE OR REPLACE PACKAGE BODY PRODUCT_PACK
AS
PROCEDURE PRODUCT_SALES(PRODUCTCODE IN
  SALES.PRODUCTNO%TYPE)  IS
------
END PRODUCT_SALES;
FUNCTION PRODUCT_PRICE(PROD_id NUMBER,PROD_amt NUMBER)
  RETURN   NUMBER IS
------
END PRODUCT_PRICE;
END PRODUCT_PACK;
```



To refer any of the member of the package we need to use:-

Package_name.Member_name

Removing Packages

- To drop the package specification and the package body,
- Syntax:

```
•DROP PACKAGE package_name;
```

- To drop the package package body
- Syntax:

```
•DROP PACKAGE BODY package_name;
```


Removing a Package

When a package is no longer required, Drop it by using the drop command.

You can drop the whole package or only the package body and retain the package specification.

Guidelines for Developing Packages

- While writing packages, see that they are as general as possible to reuse in future application
- Define the package specification before implementing the Package body.
- The package specification should contain only the types, items, and subprograms that needs to be public.
- That way other developers cannot misuse the package by basing their code on irrelevant implementation details

Guidelines for Writing Packages

Keep your packages as general as possible so that they can be reused in future applications.

Avoid writing packages that duplicate features provided by the Oracle.

Package specifications reflect the design of your application, so define them before defining the package bodies.The package specification should contain only the types, items, and subprograms that must be visible to users of the package. That way other developers cannot misuse the package by basing code on irrelevant details.

To reduce the need for recompiling when code is changed, place as few items as possible in a package spec. Changes to a package body do not require Oracle to recompile dependent procedures. However, changes to a package spec require Oracle to recompile every stored subprogram that references the package.

Advantages of Packages

- Modularity
- Easier application design
- Information Hiding
- Added Functionality
- Better Performance
- Overloading

Advantages of Using Packages

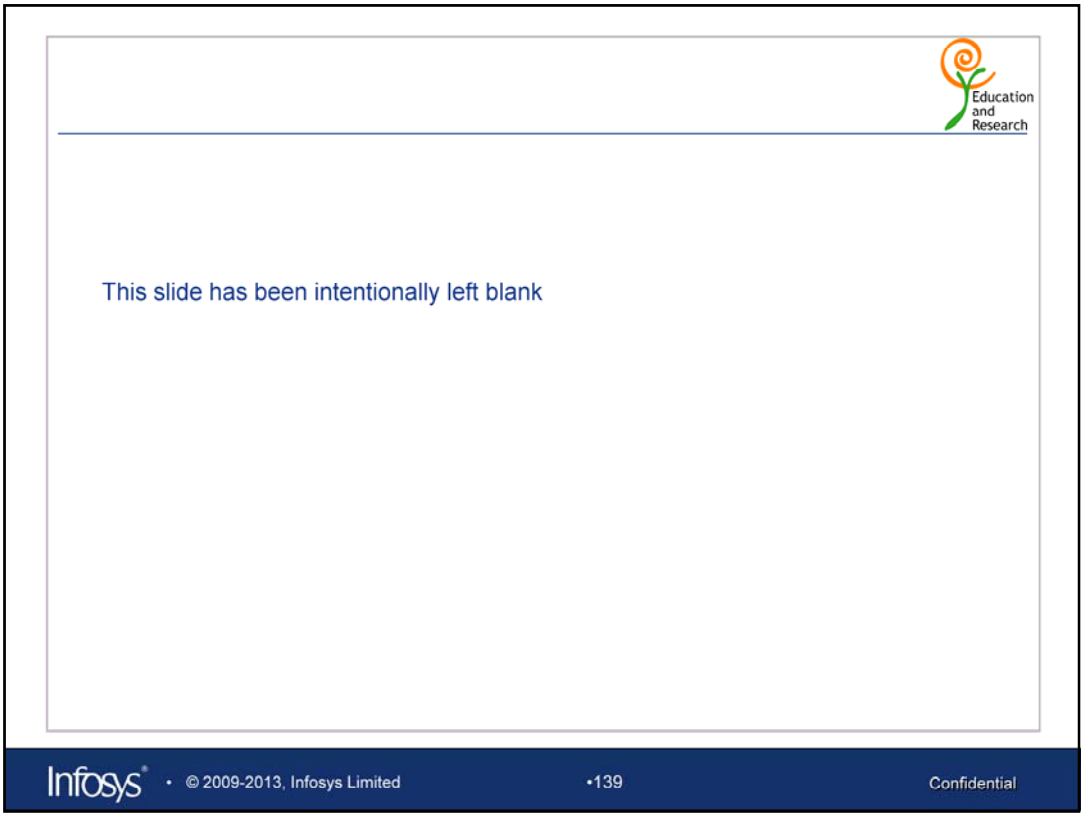
Packages provides an alternative to create procedures and functions as stand-alone schema objects, and they have various advantages.

Modularity

Packages let you encapsulate logically related types, items, and subprograms in a named PL/SQL module. Each package is easy to understand, and the interfaces between packages are simple, clear, and well defined. This aids application development

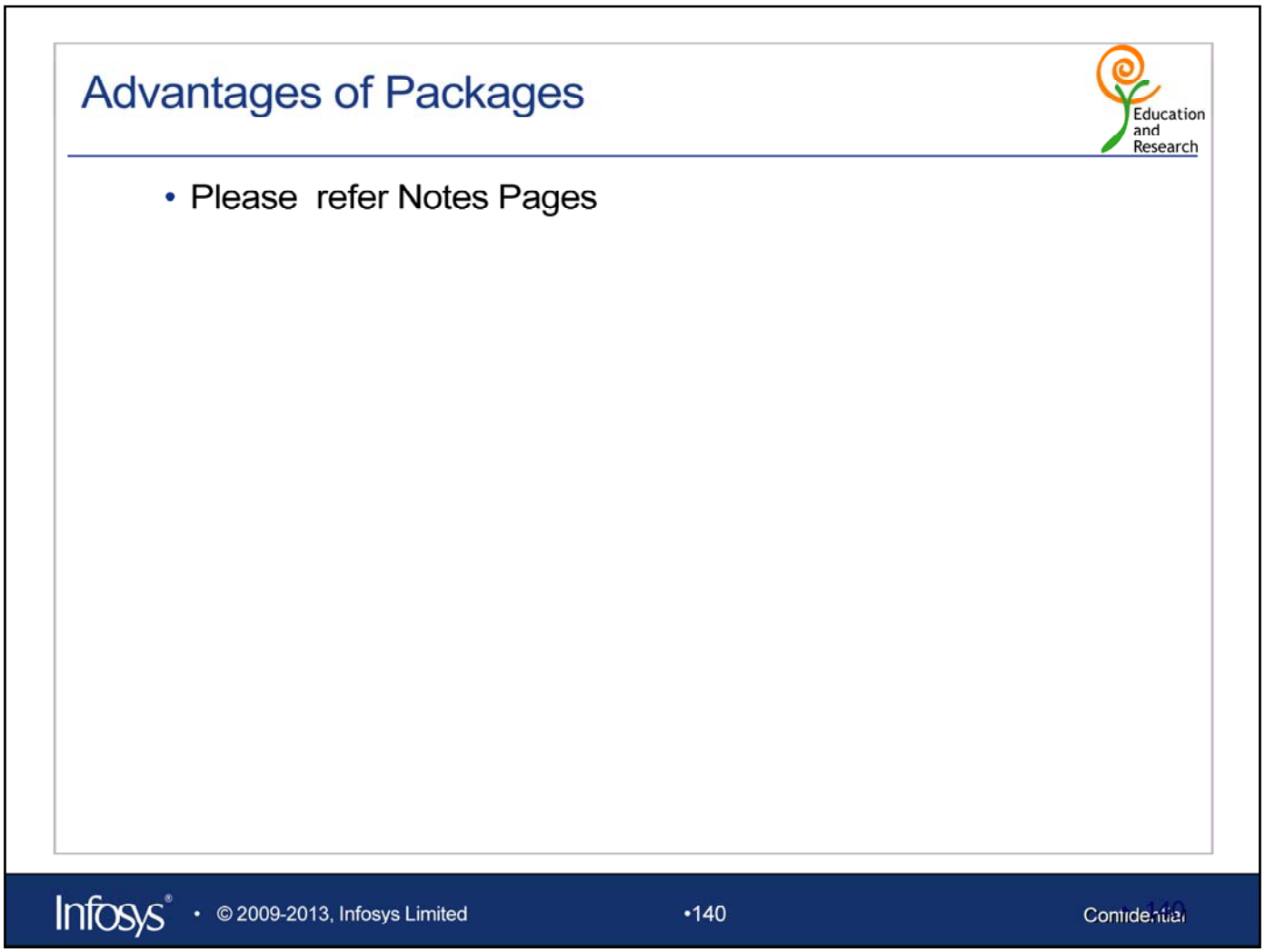
Easier Application Design

When designing an application, all you need initially is the interface information in the package specs. You can code and compile a spec without its body. Then, stored subprograms that reference the package can be compiled as well. You need not define the package bodies fully until you are ready to complete the application.



Information Hiding

With packages, you can specify which types, items, and subprograms are public (visible and accessible) or private (hidden and inaccessible). For example, if a package contains four subprograms, three might be public and one private. The package hides the implementation of the private subprogram so that only the package (not your application) is affected if the implementation changes. This simplifies maintenance and enhancement. Also, by hiding implementation details from users, you protect the integrity of the package.



Advantages of Using Packages (continued)

Added Functionality : Packaged public variables and cursors persist for the duration of a session. So, they can be shared by all subprograms that execute in the environment. Also, they allow you to maintain data across transactions without having to store it in the database.

Better Performance : When you call a packaged subprogram for the first time, the whole package is loaded into memory. So, later calls to related subprograms in the package require no disk I/O. Also, packages stop cascading dependencies and thereby avoid unnecessary recompiling. For example, if you change the implementation of a packaged function, Oracle need not recompile the calling subprograms because they do not depend on the package body.

Overloading : With packages you can overload procedures and functions, You can create multiple subprograms with the same name in the same package, which invokes depends on passing different number and type of parameters

Overloading

- PL/SQL allows two or more packaged subprograms to have the same name.
- Overloading is useful when you want a subprogram to accept similar sets of parameters that have different datatypes.
- Note: Only local or packaged subprograms can be overloaded. You cannot overload stand-alone subprograms.

Overloading

PL/SQL allows two or more packaged subprograms to have the same name.

This option is useful when you want a subprogram to accept similar sets of parameters that have different datatypes.

Note: Only local or packaged subprograms can be overloaded. You cannot overload stand-alone subprograms.

Overloading: Example


```
•CREATE or REPLACE  PACKAGE  emp_pack  AS

• function count_emp(g  IN varchar2) return number;
•  function count_emp(d  IN number) return number;

• procedure increment_sal(d IN number,sal IN number);
• procedure increment_sal(d IN number,sal  IN number,
• c IN number);
•END emp_pack;
```


Overloading: Example

The slide shows the package specification of a package with overloaded procedures and function .

The package contains two overloaded function named countemp and two overloaded procedure named incrementsal

The first declaration of function takes one parameters of type varchar2 whereas the second declaration takes one parameter of type number.

The first declaration of procedure takes two parameters of type number whereas the second declaration takes three parameters of type number.

Using Forward Declarations

You must declare identifiers before referencing them.

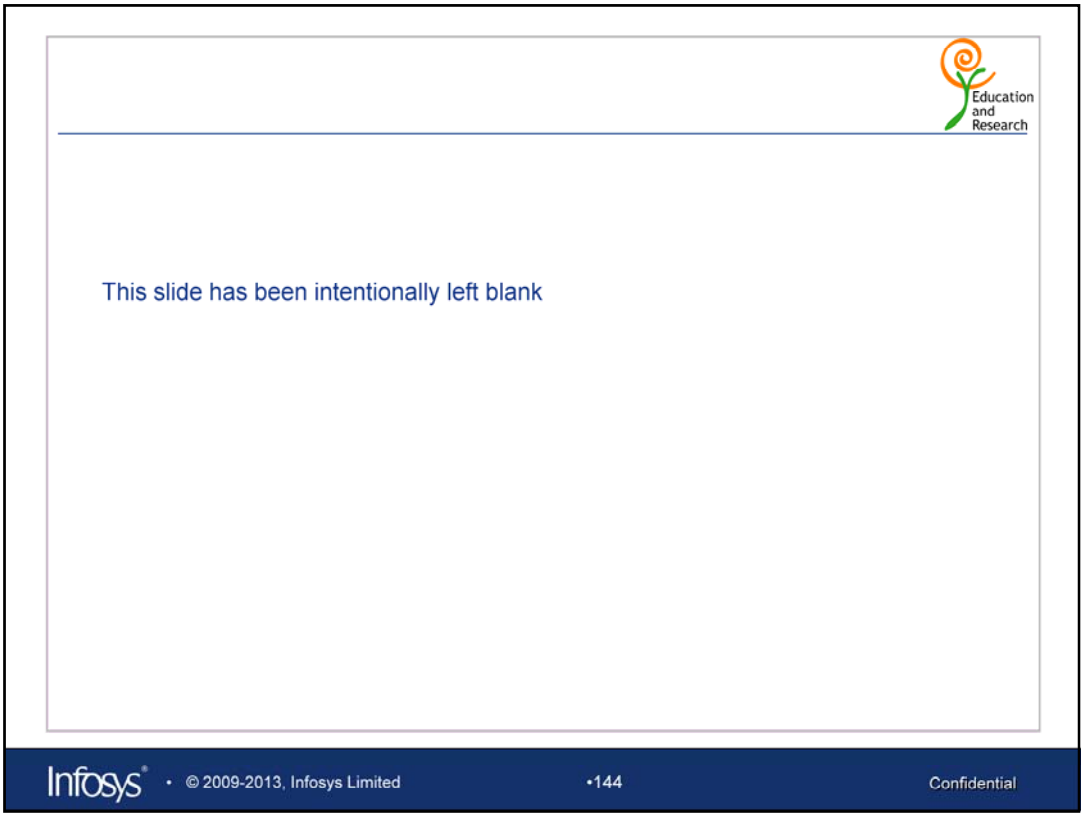
```
•CREATE OR REPLACE PACKAGE BODY pkg_forward
•IS  PROCEDURE product_sales(. . .)
•  IS  BEGIN
•   product_price(. . .);       --illegal reference
•  END;
•    PROCEDURE product_price(. . .)
•  IS  BEGIN
•    ...
•  END;

•END pkg_forward;
```


Using Forward Declarations

PL/SQL does not allow forward references. You must declare an identifier before using it. Therefore, a subprogram must be declared before calling it.

In the example given in the slide, the procedure product_price cannot be referenced because it has not yet been declared. You can solve the illegal reference problem by reversing the order of the two procedures. However, this easy solution does not always work. Suppose the procedures call each other or you absolutely want to define them in alphabetical order.

.



PL/SQL enables for a special subprogram declaration called as forward declaration. It consists of the subprogram specification terminated by a semicolon. You can use forward declarations to do the following:

- Define subprograms in logical or alphabetical order
- Define mutually recursive subprograms
- Group subprograms in a package

Mutually recursive programs are programs that call each other directly or indirectly.

Note: If you receive a compilation error that product_price is undefined, it is only a problem if product_price is a private packaged procedure. If product_price is declared in the package specification, the reference to the public procedure is resolved by the compiler

Using Forward Declarations


```
•CREATE OR REPLACE PACKAGE BODY pkg_forward
•IS
•  PROCEDURE product_price(. . .);        -- forward declaration
•  PROCEDURE product_sales(. . .)
• IS                                      -- subprograms defined
• BEGIN                                   -- in alphabetical order
•  product_price(. . .);
• . . . END;
•  PROCEDURE product_price(. . .)
• IS BEGIN
•  . . .
• END;
•END pkg_forward;
```

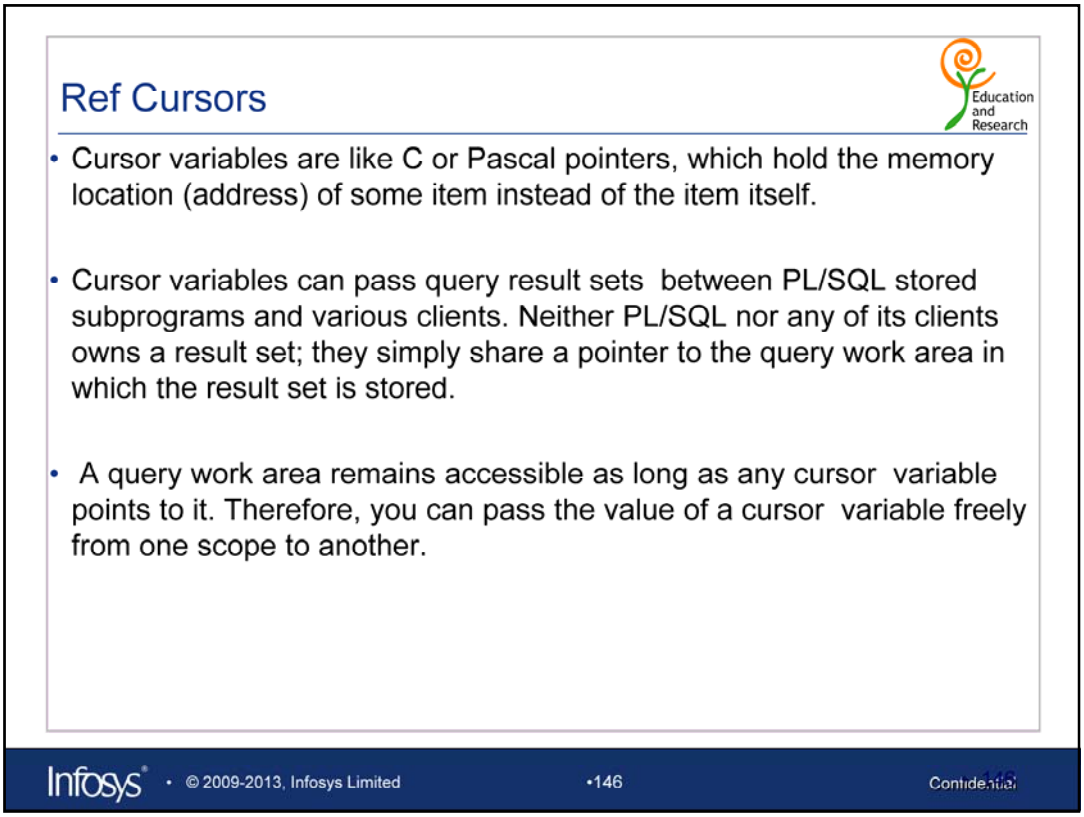

Using Forward Declarations (continued)

The formal parameter list must appear in both the forward declaration and the subprogram body.

The subprogram body can appear anywhere after the forward declaration, but both must appear in the same program unit.

Forward Declarations and Packages

Forward declarations typically let you group related subprograms in a package. The subprogram specifications go in the package specification, and the subprogram bodies go in the package body, where they are invisible to the applications. In this way, packages enable you to hide implementation details.



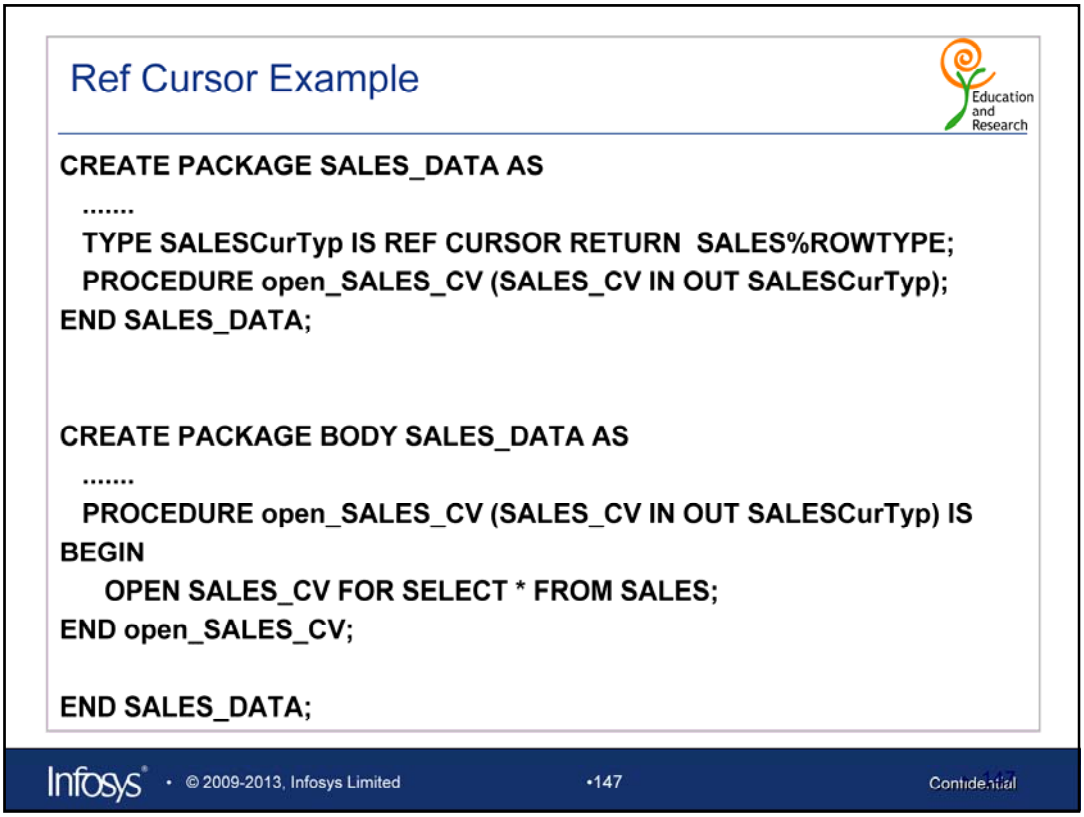
REF CURSOR

Cursor Variables : Cursor variables are like C or Pascal pointers, which hold the memory location (address) of some item instead of the item itself. So, declaring a cursor variable creates a pointer, *not* an item. In PL/SQL, a pointer has datatype REF X, where REF is short for REFERENCE and X stands for a class of objects. 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. Whereas a cursor always refers to the same query work area, 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.

Why Use Cursor Variables?

Mainly, you use cursor variables 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

A query work area remains accessible as long as any cursor variable



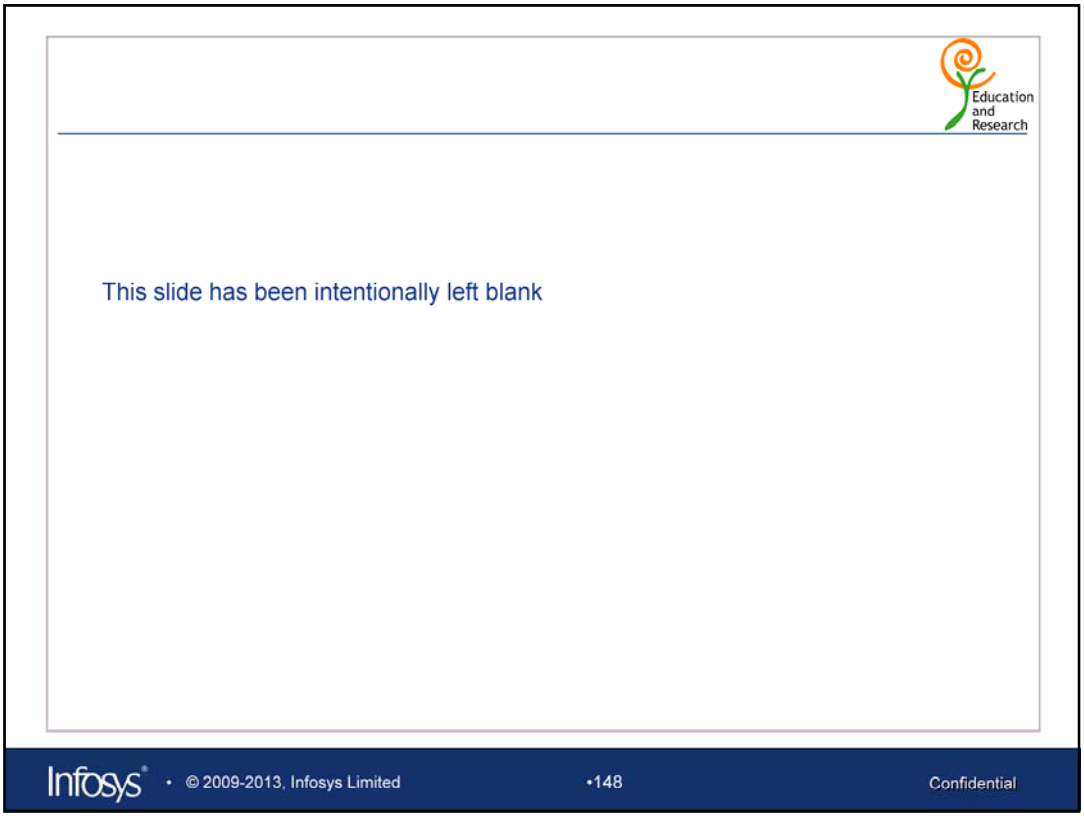
REF CURSOR Types

To create cursor variables, you take two steps. First, you define a REF CURSOR type, then declare cursor variables of that type. You can define REF CURSOR types in any PL/SQL block, subprogram, or package .

Typically, you open a cursor variable by passing it to a stored procedure that declares a cursor variable as one of its formal parameters.

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.

Alternatively, you can use a standalone procedure to open the cursor variable. Simply define the REF CURSOR type in a separate package, then reference that type in the standalone procedure.

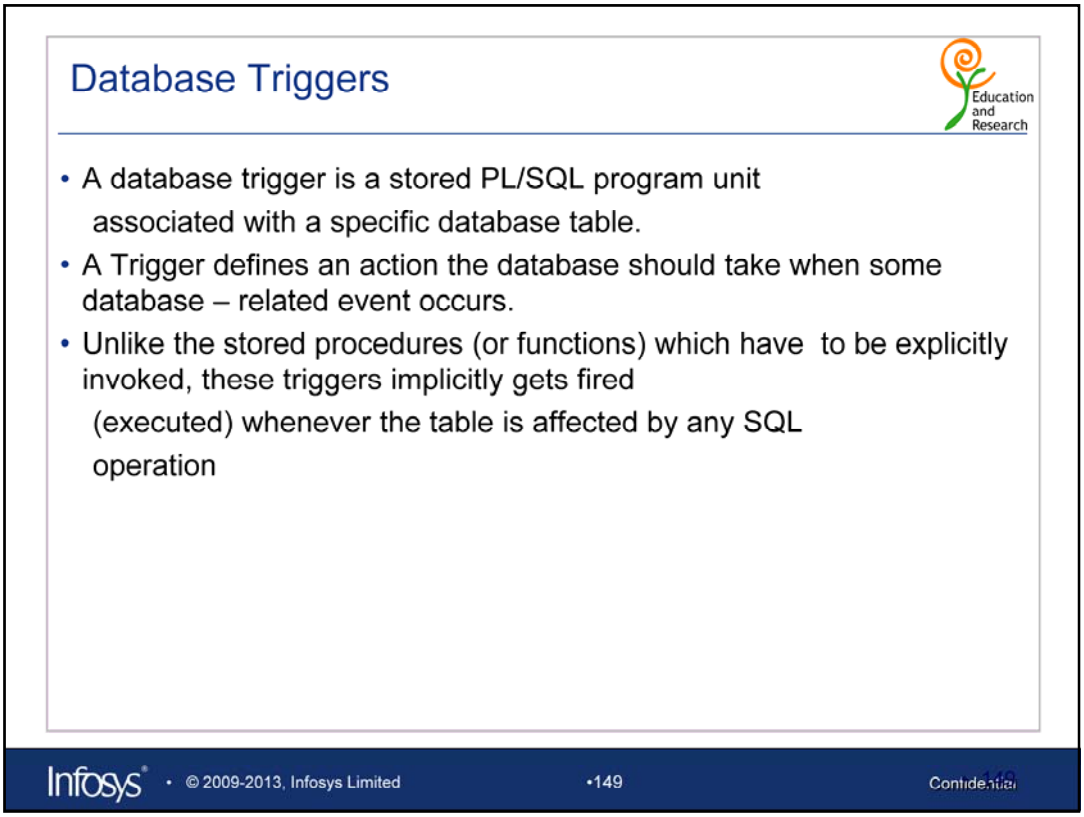


Fetching from a Cursor Variable

The FETCH statement retrieves rows from the result set of a multi-row query.

In the following example, you fetch rows one at a time from the cursor variable SALES_CV into the user-defined record SALES_rec:

```
LOOP
  /* Fetch from cursor variable. */
  FETCH emp_cv INTO SALES_rec;
  EXIT WHEN SALES_CV%NOTFOUND; -- exit when last row is fetched
  -- process data record
END LOOP;
CLOSE SALES_CV;
```



A database trigger is a PL/SQL block that can be associated with a specific database table.

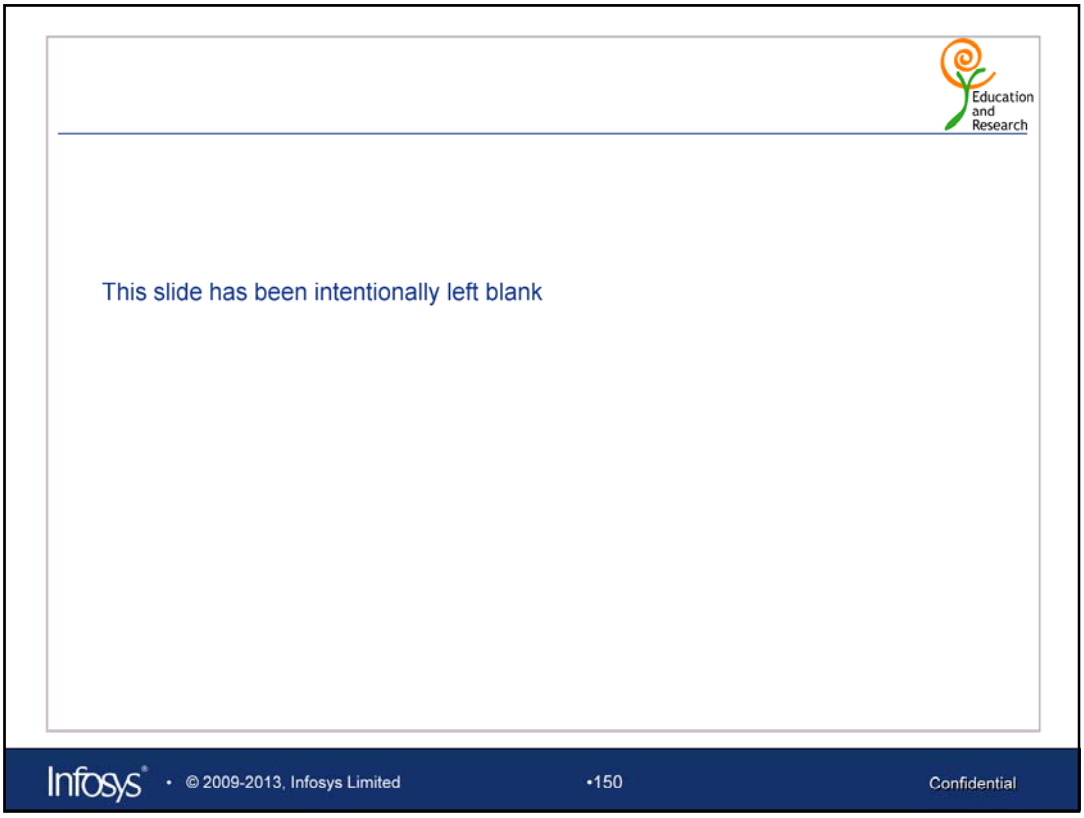
Triggers are programs written in Pl/SQL that execute automatically whenever a table or view is modified or when some user action or database system actions occur.

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

The purpose is to perform a specific service when a specified operation occurs on a table.

You create a database trigger by specifying a database table and by specifying that before or after a database operation(INSERT, UPDATE, DELETE) on the table, a procedure is to be invoked. When the specified operation occurs on the table, ORACLE automatically ‘fires’ the database trigger.

Applications where Database Triggers are useful



To verify data integrity on insertion or update
Customizing database
Implement delete cascades
Log events transparently
To enforce referential integrity
To enforce complex business rules
To provide sophisticated auditing
Initiate business processes
Derive column values automatically
To enforce complex security authorizations
Maintain replicated data
To gather statistics in table access
To prevent invalid transactions

Parts of a Trigger

- A database trigger has three basic parts:
  - Triggering event
  - Trigger constraint (Optional)
  - Trigger action

Syntax for creating a Database Trigger


```
CREATE [OR REPLACE] TRIGGER  <trigger-name> BEFORE |
  AFTER | INSTEAD OF
      .
      .
DELETE | [OR] INSERT | [OR] UPDATE [ OF <column> [,
  <column>...]] ON <table>
      .
[ FOR EACH ROW [ WHEN  <condition>] ]
BEGIN
/* PL/SQL Block */
      .
...   .
END;
```

- •Triggering
- •Event
- •Triggering
- •Restriction
- •Triggering
- •Action

Types of Triggers

•Name	•Statement level Trigger	• Row level Trigger •(With FOR EACH ROW option
•BEFORE	•Oracle fires the trigger only •once, before executing the •triggering statement	• Oracle fires the trigger before • modifying each row affected by • the triggering statement
•AFTER	•Oracle fires the trigger only •once, after executing the •triggering statement	• Oracle fires the trigger after • modifying each row affected by • the triggering statement
•INSTEAD OF	•Oracle fires the trigger only • once, to do something else •instead of performing the action •that executed the trigger.	• Oracle fires the trigger for each • row, to do something else • instead of performing the action • that executed the trigger.

•*Source : Created by author for Infosys Technologies*

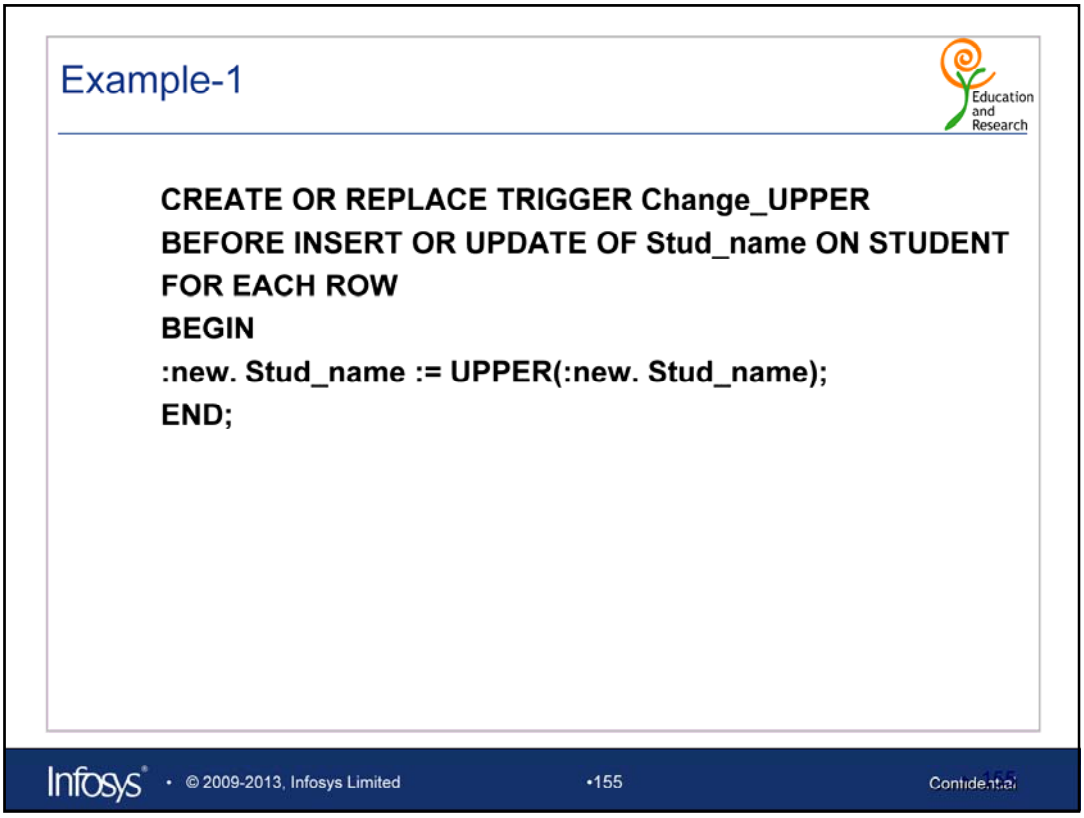
Guidelines for Designing Triggers

- **Create triggers:**
- When you need to perform related actions
- When you wanted to perform centralize global operations

- **Do not create triggers:**
- Where functionality is already existing, since duplication will increase
- You can create stored procedures and invoke them in a trigger, if your PL/SQL code is very lengthy.
- The excessive use of triggers increase more complexity

Create triggers:
When you need to perform related actions
When you wanted to perform centralize global operations

Do not create triggers:
Where functionality is already existing since duplication will increase
You can create stored procedures and invoke them in a trigger, if your PL/SQL code is very lengthy.
The excessive use of triggers increase more complexity



The trigger in the PPT converts all student names into upper case whenever a user inserts a row into the STUDENT table or updates this column

Using OLD and NEW Qualifiers

Within a ROW trigger, reference the value of a column before and after the data change by prefixing it with the OLD and NEW qualifier.

The OLD and NEW qualifiers are available only in ROW level triggers.

:OLD Qualifier will result in NULL value for Insert statement and :NEW qualifier will lead to NULL for DELETE statement.

Prefix these qualifiers with a colon (:) in every SQL and PL/SQL statement.

There is no colon (:) prefix if the qualifiers are referenced in the WHEN restricting condition.

Note: Row level triggers can decrease the performance if you do a lot of updates on larger tables.

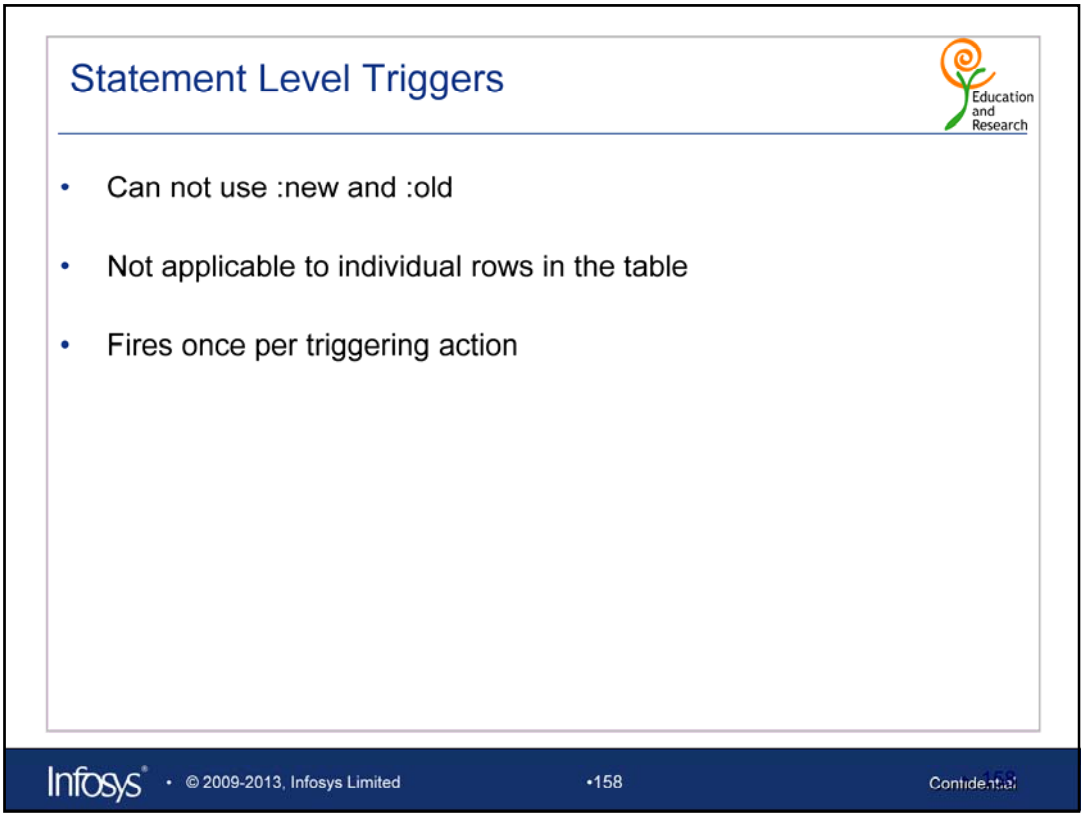
Example-2

```
CREATE OR REPLACE TRIGGER total_price
AFTER DELETE OR INSERT OR UPDATE OF productno  ON SALES FOR
  EACH ROW

BEGIN

IF DELETING
   UPDATE  PRODUCT_COUNT SET totprice=totprice - :OLD.prodprice
   WHERE productno = :OLD.productno;
END IF;

IF INSERTING THEN
   UPDATE PRODUCT_COUNT  SET totprice=totprice + :NEW.prodprice
   WHERE productno =  :NEW. productno;
END IF;
```


The trigger is written to update a PRODUCT_COUNT , whenever a row is inserted, updated or deleted from SALES table.

Example-2(Contd..)


```
IF UPDATING THEN
    UPDATE PRODUCT_COUNT  SET totprice = totprice - :OLD.prodprice
    WHERE productno = :OLD.productno;

    UPDATE PRODUCT_COUNT SET totprice = totprice + :NEW.prodprice
    WHERE productno = :NEW.productno;
END IF;


END;
```



Creating Row or Statement Triggers

Create a statement trigger or a row trigger based on the requirement that the trigger must fire once for each row affected by the triggering statement, or just once for the triggering statement, regardless of the number of rows affected.

When the triggering data manipulation statement affects a single row, both the statement trigger and the row trigger fire exactly once.

Statement Level Trigger

Education and Research

•Example
:

```
•CREATE OR REPLACE TRIGGER trig1 BEFORE INSERT ON SALES

• BEGIN
•  IF (TO_CHAR(SYSDATE,'DY') IN ('SAT','SUN')) OR
•     (TO_CHAR(SYSDATE,'HH24') NOT BETWEEN '10' AND '17')
•  THEN RAISE_APPLICATION_ERROR (-20112,'You are not allowed
to do any transaction after office hours.');

•  END IF;
•END;
```


Creating DML Statement Triggers

You can create a BEFORE statement trigger in order to prevent the triggering operation from succeeding if a certain condition is violated.

For example, create a trigger to restrict inserts into the SALES table to certain business hours, Monday through Friday.

If a user attempts to insert a row into the SALES table on Saturday, the user sees the message, the trigger fails, and the triggering statement is rolled back. Remember that the RAISE_APPLICATION_ERROR is a server-side built-in procedure that returns an error to the user and causes the PL/SQL block to fail.

When a database trigger fails, the triggering statement is automatically rolled back by the Oracle
server.

Mutating Error

- A table is called mutating when it is used in any DML statements
- Querying a mutating table with in the trigger leads to mutation error

Example below will lead to mutation error:

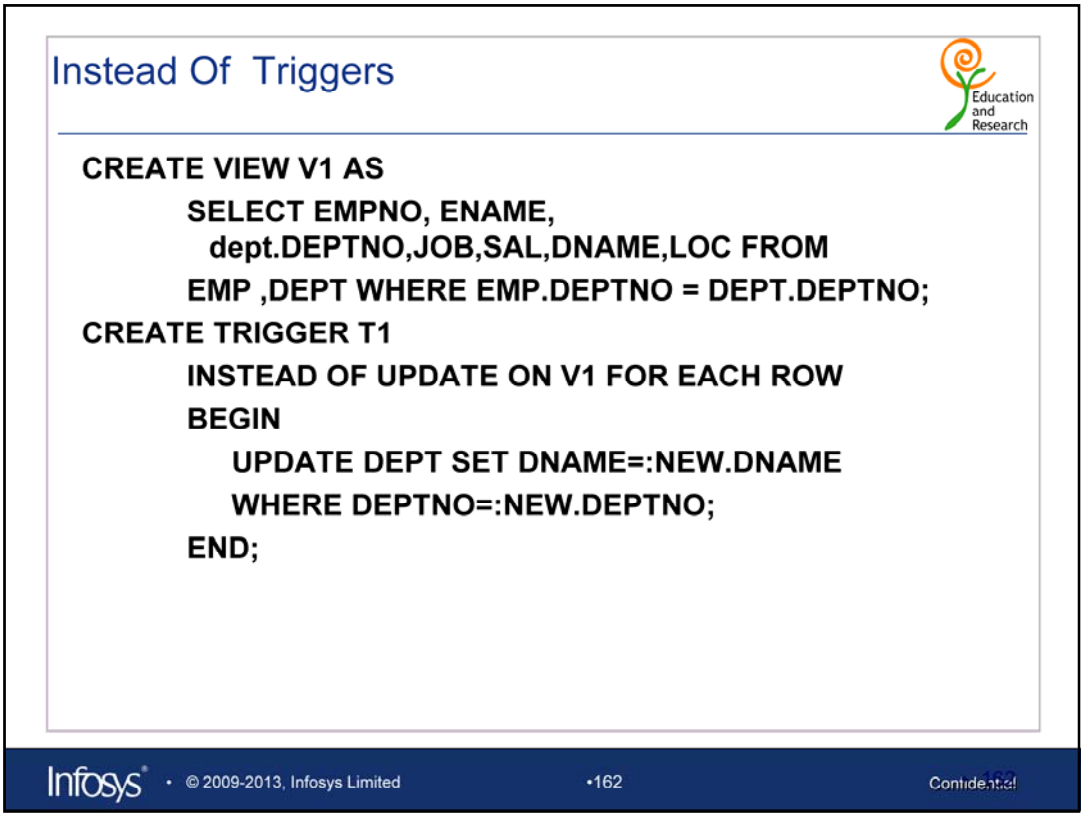
```
CREATE OR REPLACE TRIGGER trig1
        AFTER  INSERT ON STUDENT
        FOR EACH ROW

        BEGIN
        UPDATE STUDENT SET .....
        END;
```

Mutating Error

Education and Research

Exceptions

- Before Insert trigger do not lead to mutation error
- Statement level trigger do not lead to mutation error



Instead of triggers are written only on views to achieve join view updatability.

INSTEAD OF is a valid clause only for views. You cannot specify an INSTEAD OF trigger on a table.

If a view has INSTEAD OF triggers, any views created on it must have INSTEAD OF triggers, even if the views are inherently updatable.

When defining INSTEAD OF triggers for LOB columns, you can read both the :OLD and the :NEW value, but you cannot write either the :OLD or the :NEW values.

A trigger is written in the PPT to update the DEPT table when ever a user tries to update using a view

Dropping Triggers

Drop a trigger

```
DROP TRIGGER <Trigger_Name>;
```

Disable/Enable a trigger

```
ALTER TRIGGER <Trigger_Name> DISABLE/ENABLE;
```

Data Dictionary Table

```
USER_TRIGGERS;
```

•View the Code of Triggers


```
•SELECT trigger_name, trigger_type, triggering_event,
•       table_name, referencing_names,
•       status, trigger_body
•FROM   user_triggers
•WHERE  trigger_name = 'Update_Trig';
```


Example

Use the USER_TRIGGERS data dictionary view to display information about the specified trigger.

•Difference Between Database Triggers and Stored Procedures/Functions

•Triggers	•Procedures/Function
•Defined with CREATE Trigger	•Defined with CREATE Procedure/Function
•USER_TRIGGERS Data dictionary contains source code	•USER_SOURCE Data dictionary contains source code
•Implicitly invoked	•Explicitly invoked
•COMMIT, SAVEPOINT, and ROLLBACK cannot be used.	•COMMIT, SAVEPOINT, and ROLLBACK can be used.

Triggers are fully compiled when the CREATE TRIGGER command is issued and the P code is stored in the data dictionary.

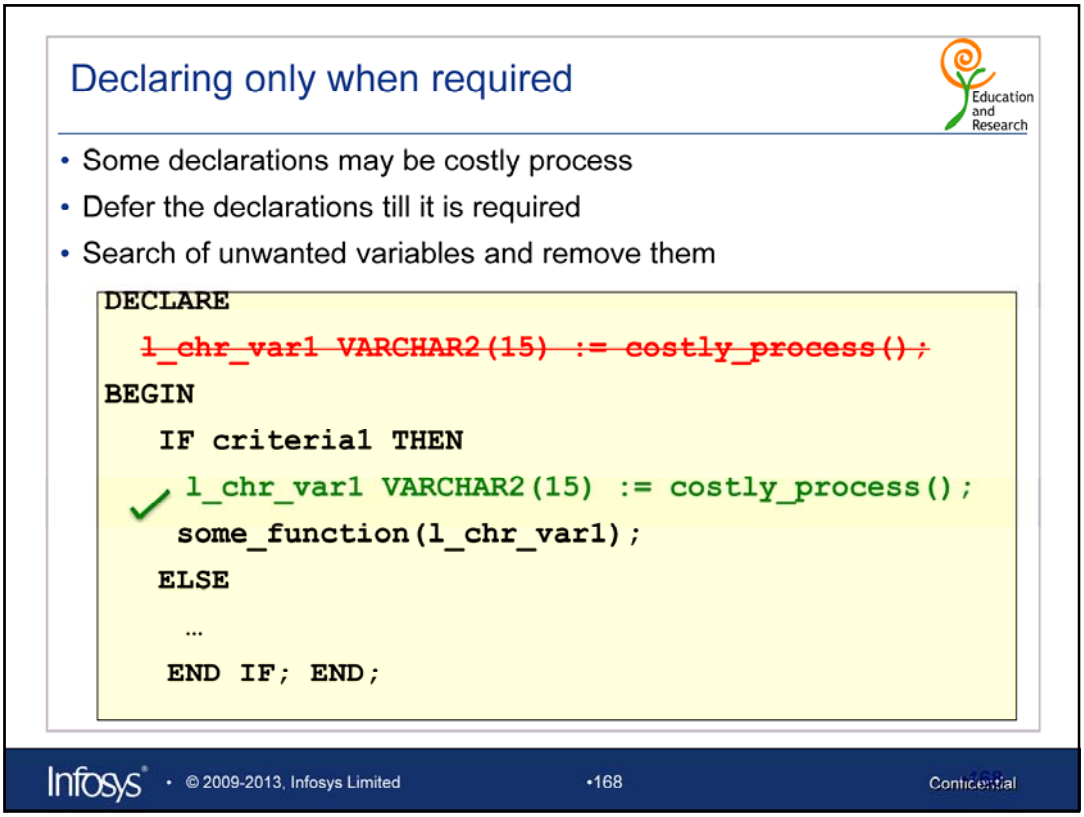
If errors occur during the compilation of a trigger, the trigger is still created.

Database Trigger	Stored Procedure
Invoked implicitly	Invoked explicitly
COMMIT, ROLLBACK, and SAVEPOINT statements are not allowed within the trigger body. It is possible to commit or rollback indirectly by calling a procedure, but it is not recommended because of side effects to transactions.	COMMIT, ROLLBACK, and SAVEPOINT statements are permitted within the procedure body.

PL/SQL Best Practices

PL/SQL Best Practices

- PL/SQL code may sometimes use excessive CPU cycles even when there is no database access. This degrades performance
- Performance can be improved by adopting proven best practices while coding the application

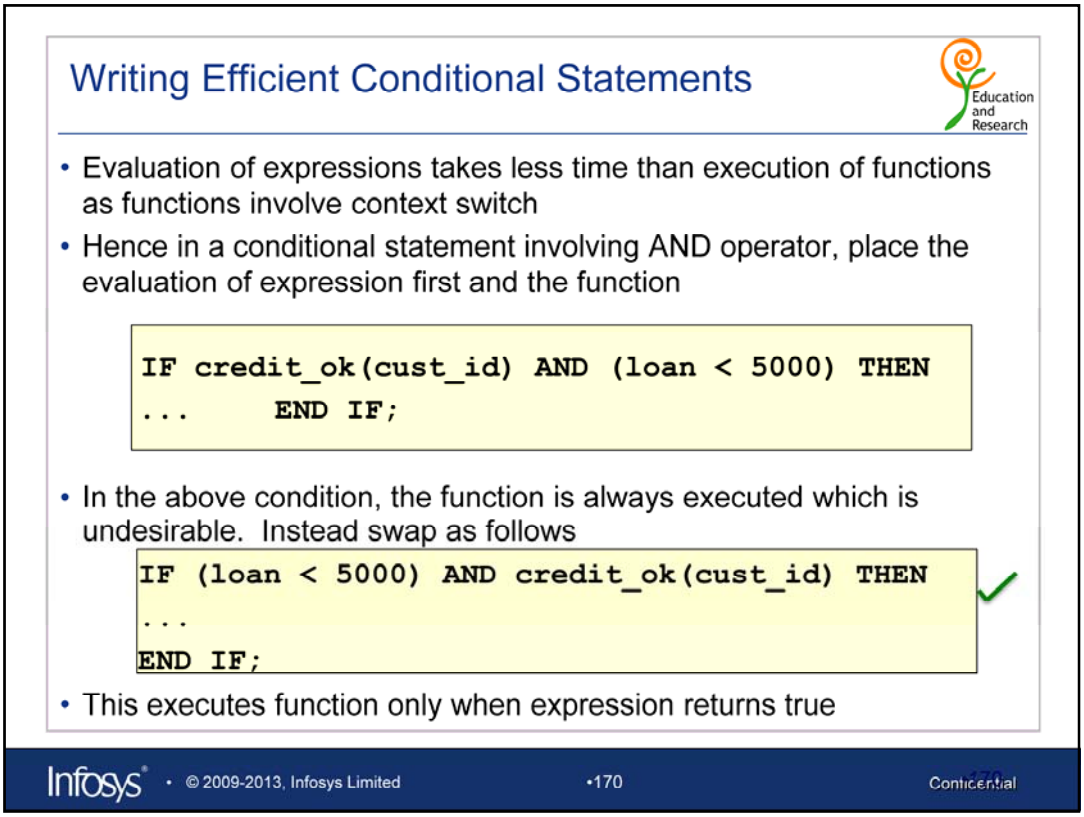


Declaring only when required

Some of the declarations may need a costly process to assign a value for initialization. Hence it is recommended that always defer the declarations of variables till it is required. It is also advised to search for unwanted variables and remove them from the program.

Prefer Built-in Functions over the user defined

- Do not try to implement the functionalities which are already provided by built-in functions.
- Use the built-in functions wherever required as they are more optimized than the user-defined functions



Conditional Statement

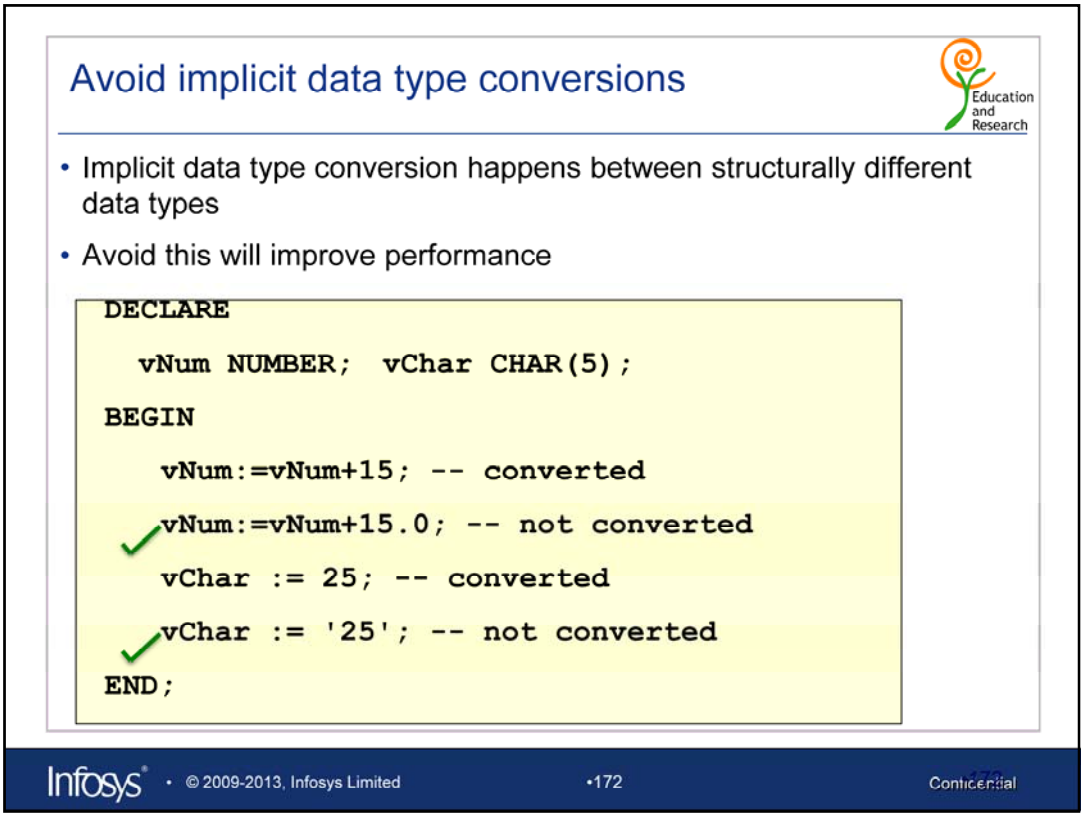
If the conditional statement contains the AND operator, then the first operand of AND is always executed and second operand of AND is executed only when the first operand evaluates to true. Hence it is desirable to place expressions which consume less time, first and the functions taking more time, second. This will execute the function only when the first operand which is expression, evaluates to true.

Check the Loop Statements

- Always check if loop contains only those statements which depend on the loop
- In the following code, UPPER is executed unnecessarily for every iteration.

```
BEGIN
     FOR temp_rec IN temp_cur
     LOOP
        Process_new_procedure(UPPER(variable1));
     END LOOP
END;
```

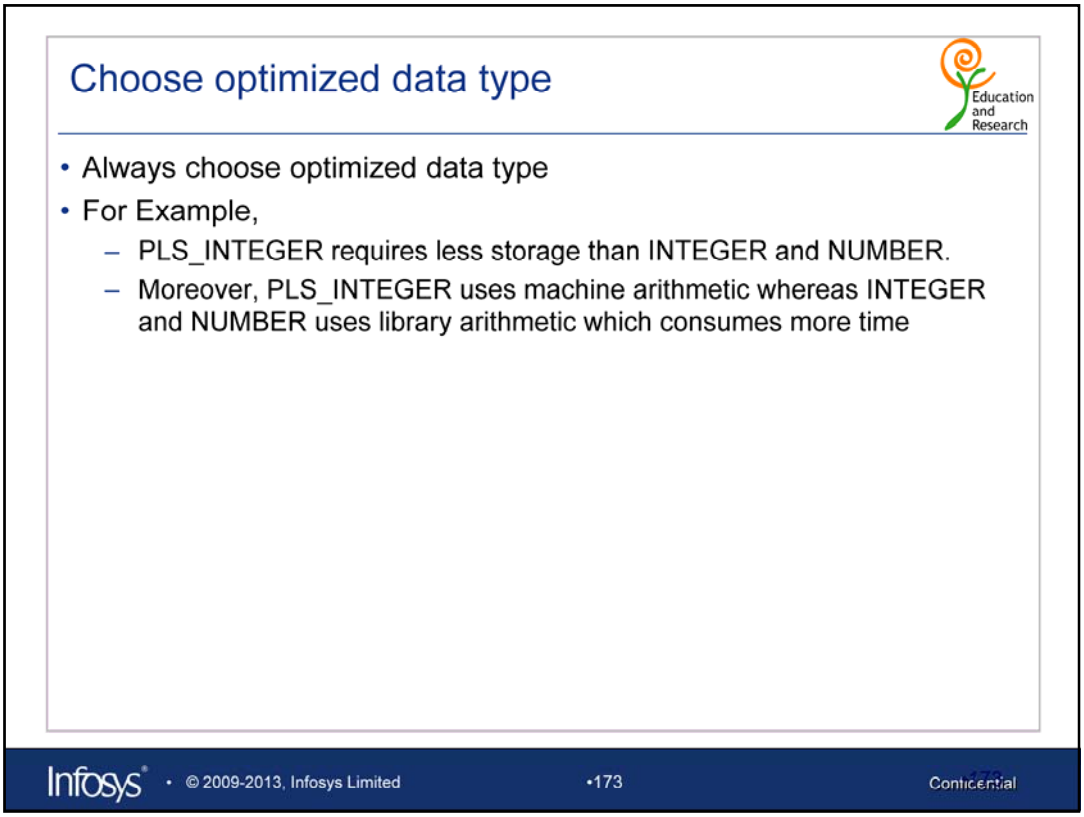
- Exit the loop as soon as the required job is done



Avoid implicit data type conversions

PL/SQL implicitly converts structurally different data types in an assignment statement. For example, assigning a PLS_INTEGER variable to a NUMBER variable results in an implicit conversion because their internal representations are different. Hence it is always recommended to assign values to variables whose data types are compatible. This will improve performance.

In the example shown in the slide, the integer literal 15 is represented internally as a signed 4-byte quantity, so PL/SQL must convert it to an Oracle number before the addition. However, the floating-point literal 15.0 is represented as a 22-byte Oracle number, so no conversion is necessary.

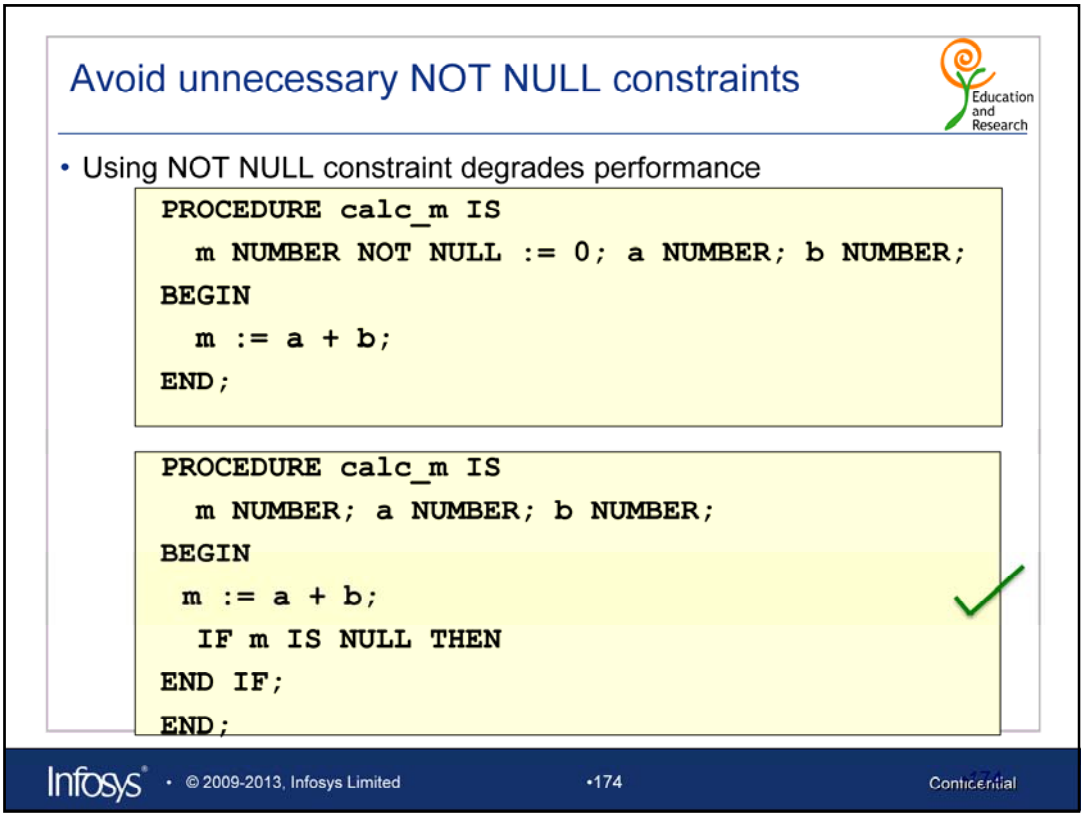


Choose optimized data type

Always choose an optimized data type.

For example,

the NUMBER data type which is 22 byte is good at supporting portability but may degrade the performance. Hence whenever a integer variable is to be declared, use PLS_INTEGER as it requires less storage that INTEGER or NUMBER data types. Moreover, PLS_INTEGER used machine arithmetic, whereas, INTEGER and NUMBER uses library arithmetic which consumes more time.



Avoid unnecessary NOT NULL constraints

Using NOT NULL constraint degrades the performance. Hence minimize the use of NOT NULL constraint.

In the example shown in the slide, since m is constrained by NOT NULL, the value of the expression a + b is assigned to a temporary variable, which is then tested for nullity. If the variable is not null, its value is assigned to m. Otherwise, an exception is raised. However, if m were not constrained, the value would be assigned to m directly. Hence check for NULL programmatically and then take necessary action to improve performance.

Summary

- Subprograms are stored in the database
- Procedures and functions
- IN , OUT and IN OUT Parameters
- Temporary Procedures and Functions
- Packages
- Database Triggers
- Instead of Triggers

Summary

- PL/SQL provides the procedural capability through various programming constructs
- Collections are single-dimensional structures containing homogeneous elements
- FORALL and BULK COLLECT help in bulk binding to avoid multiple context switches
- Cursors give access to active result set to manipulate them row by row
- Exceptions can be handled using exception handlers written in the exception section of the block
- Subprograms are stored in the database
- Procedures and functions
- IN , OUT and IN OUT Parameters
- Temporary Procedures and Functions
- Packages
- Database Triggers
- Instead of Triggers
- Following PL/SQL best practices will improve the performance of the code

References

- Oracle Documentation, *"Oracle 9i Database Concepts, Release 2 (9.2)"*, Oracle Corporation, March 2002
- Oracle Documentation, "Oracle 9i SQL Reference, Release 2 (9.2)", Oracle Corporation, March 2002
- Oracle Documentation, "PL/SQL User's Guide and Reference, Release 9.0.1", Oracle Corporation, June 2001
- Kevin Loney and George Koch, *"Oracle 9i: The Complete Reference"*, Oracle Press, 2002

Thank You