



Overview of PL/SQL





Course Objectives

After completing this course, you should be able to do the following:

- **Describe the purpose of PL/SQL**
- **Describe the use of PL/SQL for the developer as well as the DBA**
- **Explain the benefits of PL/SQL**
- **Create, execute, and maintain procedures, functions, packages, and database triggers**
- **Manage PL/SQL subprograms and triggers**
- **Describe Oracle supplied packages**
- **Manipulate large objects (LOBs)**

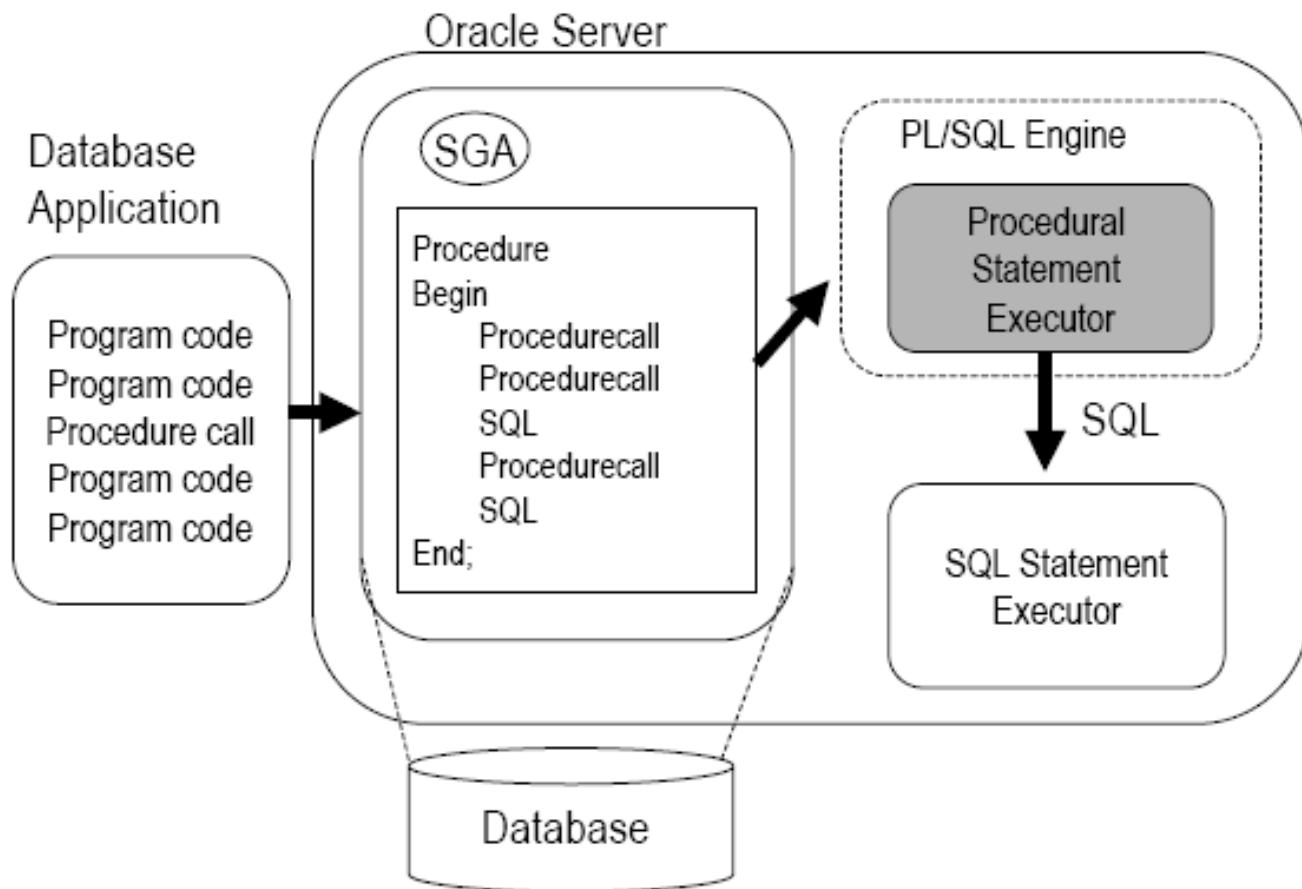


About PL/SQL

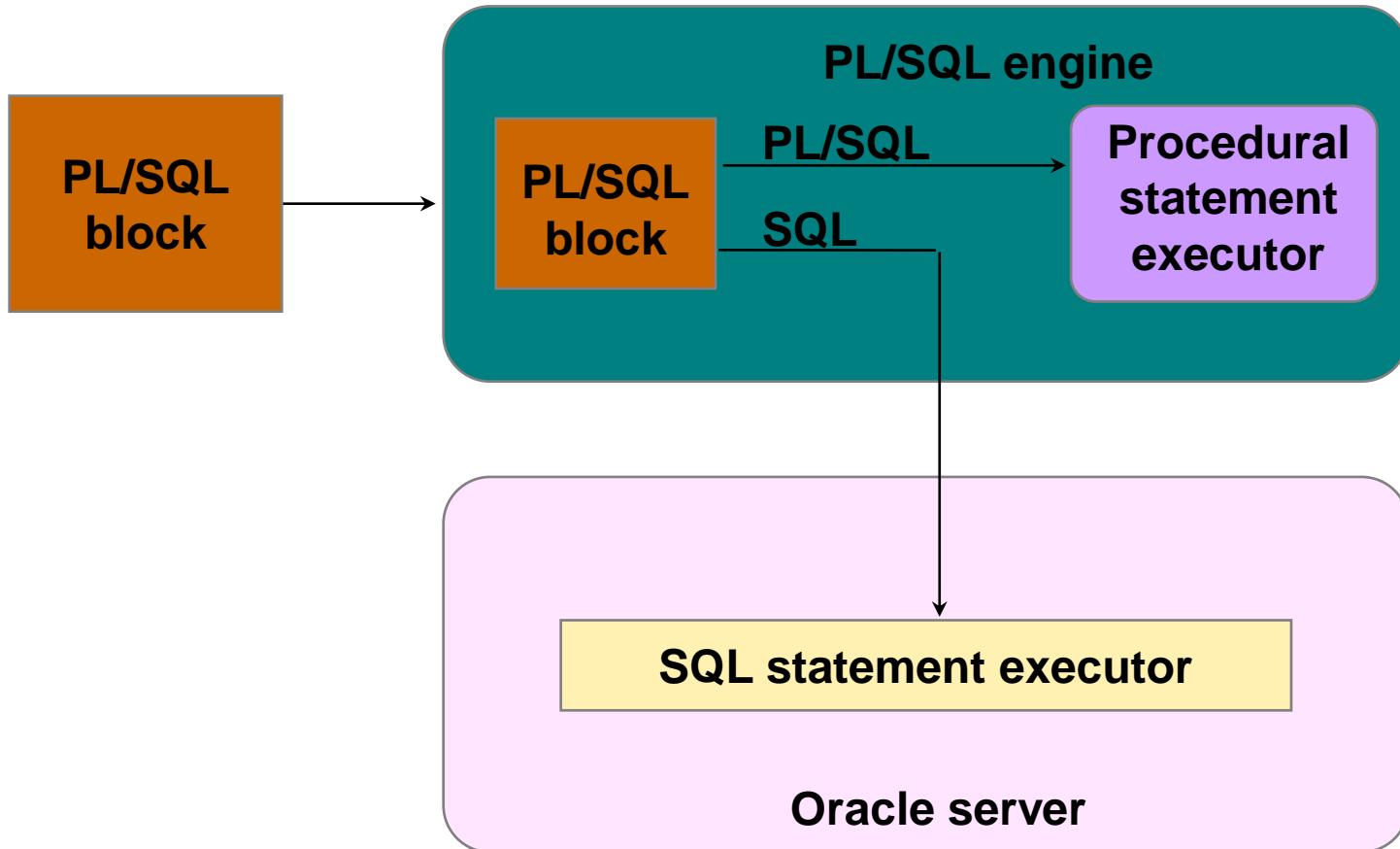
- **PL/SQL is the procedural extension to SQL with design features of programming languages.**
- **Data manipulation and query statements of SQL are included within procedural units of code.**

PL/SQL Improves Performance

Architecture

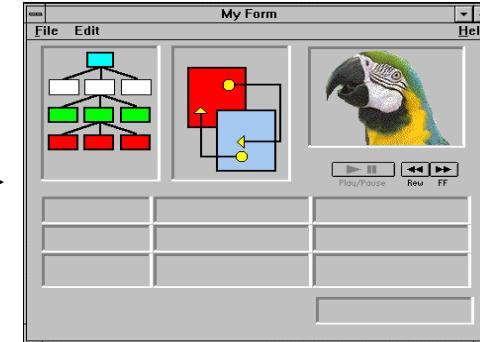


PL/SQL Environment

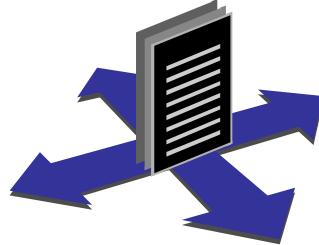


Benefits of PL/SQL

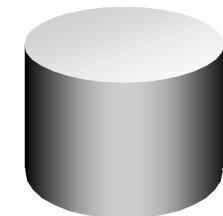
Integration



Application



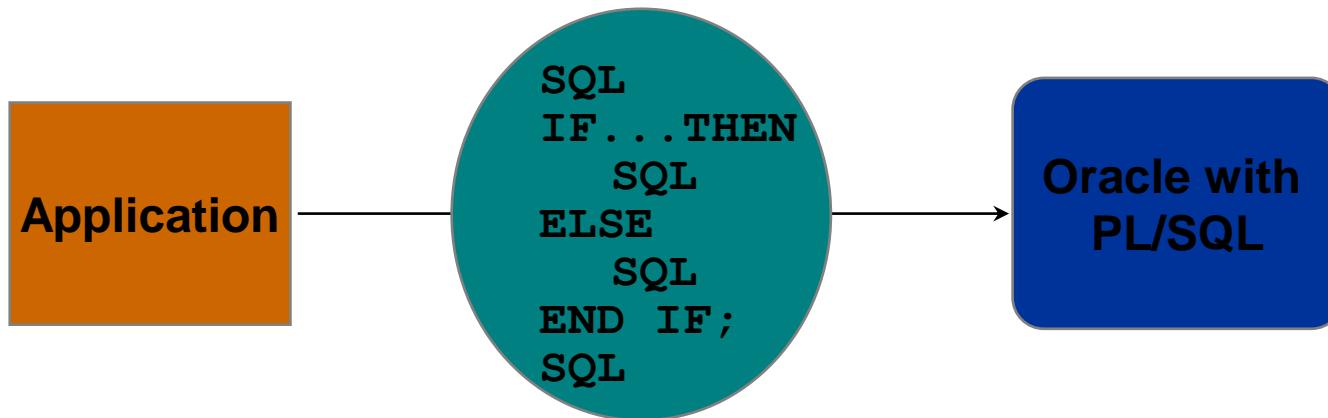
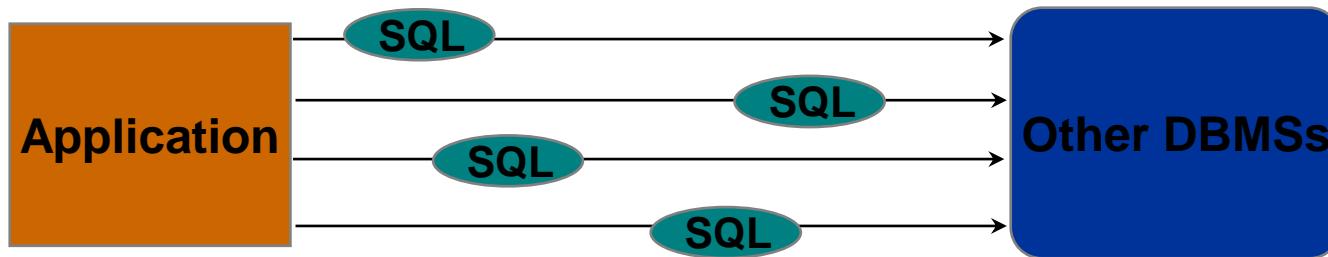
Shared
library



Oracle server

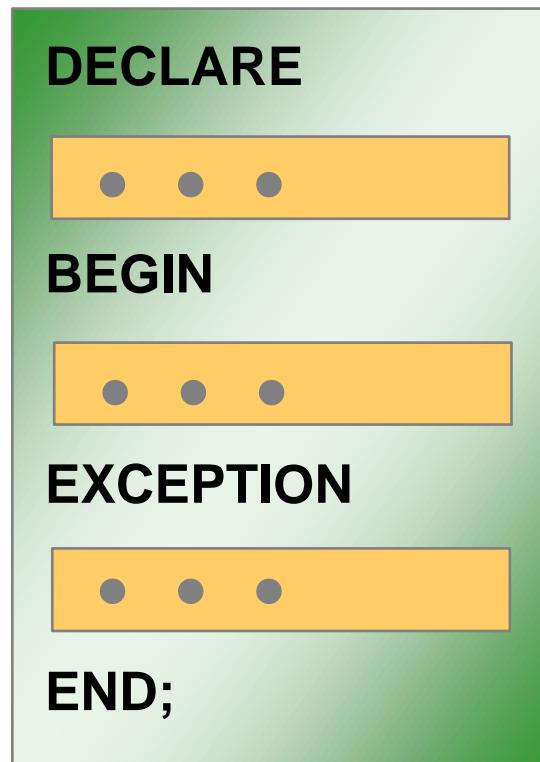
Benefits of PL/SQL

Improved Performance



Benefits of PL/SQL

Modularize program development



PL/SQL Block

- **PL/SQL is a block structured language**
- **It is composed of one or more blocks**
- **Types of Blocks**
 - **Anonymous Blocks:** constructed dynamically and executed only once
 - **Named Blocks:** Subprograms, Triggers, etc.
 - **Subprograms:** are named PL/SQL blocks that are stored in the database and can be invoked explicitly as and when required; e.g. Stored Procedures, Stored Functions and Packages
 - **Triggers:** are named blocks that are also stored in the database; Invoked implicitly whenever the triggering event occurs; e.g. Database Triggers



PL/SQL Block Structure

- A PL/SQL block has the following structure

DECLARE (Optional Declaration Section)
Variables, constants, types, cursors,
user-defined exceptions, etc.

BEGIN (Mandatory Executable/Business Logic section)

SQL and PL/SQL statements

EXCEPTION (Optional Exception-handling section)

Trapping Errors occurred in executable section

END;

PL/SQL Block Structure

- **PL/SQL Block consists of three sections**

- **Declarative:** Contains declarations of variables, constants, cursors, user-defined exceptions and types (Optional)
- **Executable:** Contains SQL statements to manipulate data in the database and PL/SQL statements to manipulate data in the block
- **Exception Handling:** Specifies the actions to perform when errors and abnormal conditions arise in the executable section (Optional)



Benefits of PL/SQL

- **PL/SQL is portable.**
- **You can declare variables.**



Benefits of PL/SQL

- You can program with procedural language control structures.
- PL/SQL can handle errors.



Benefits of Subprograms

- **Easy maintenance**
- **Improved data security and integrity**
- **Improved performance**
- **Improved code clarity**

Invoking Stored Procedures and Functions



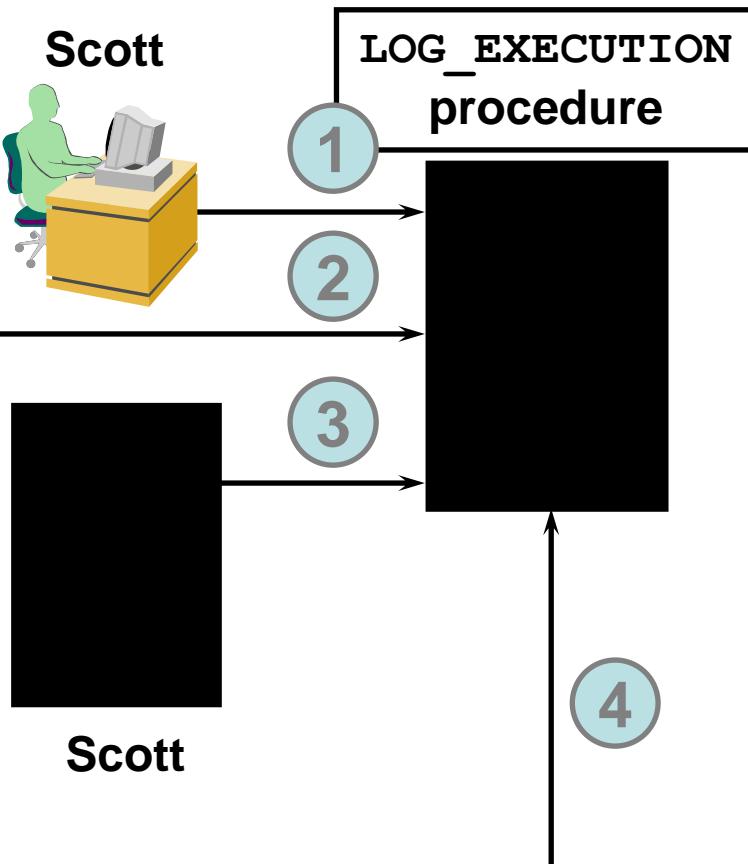
Oracle
Portal



Oracle
Discoverer



Oracle
Forms
Developer





Summary

- PL/SQL is an extension to SQL.
- Blocks of PL/SQL code are passed to and processed by a PL/SQL engine.
- Benefits of PL/SQL:
 - Integration
 - Improved performance
 - Portability
 - Modularity of program development
- Subprograms are named PL/SQL blocks, declared as either procedures or functions.
- You can invoke subprograms from different environments.

Declaring Variables

A dark blue circle containing the text 'www.' in white.A light purple circle containing the text '.com' in dark purple.



Objectives

After completing this lesson, you should be able to do the following:

- **Recognize the basic PL/SQL block and its sections**
- **Describe the significance of variables in PL/SQL**
- **Declare PL/SQL variables**
- **Execute a PL/SQL block**

PL/SQL Block Structure

DECLARE (Optional)

Variables, cursors, user-defined exceptions

BEGIN (Mandatory)

- SQL statements
- PL/SQL statements

EXCEPTION (Optional)

Actions to perform when errors occur

END ; (Mandatory)



Executing Statements and PL/SQL Blocks

```
DECLARE
    v_variable  VARCHAR2(5);
BEGIN
    SELECT column_name
    INTO v_variable
    FROM table_name;
EXCEPTION
    WHEN exception_name THEN
        ...
END;
```

```
DECLARE
    ...
BEGIN
    ...
EXCEPTION
    ...
END;
```

Block Types

Anonymous

```
[DECLARE]  
  
BEGIN  
    --statements  
  
[EXCEPTION]  
  
END ;
```

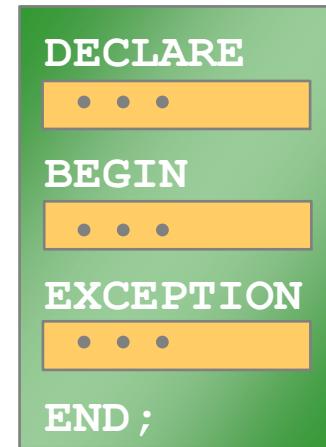
Procedure

```
PROCEDURE name  
IS  
  
BEGIN  
    --statements  
  
[EXCEPTION]  
  
END ;
```

Function

```
FUNCTION name  
RETURN datatype  
IS  
  
BEGIN  
    --statements  
    RETURN value;  
  
[EXCEPTION]  
  
END ;
```

Program Constructs





Use of Variables

Variables can be used for:

- **Temporary storage of data**
- **Manipulation of stored values**
- **Reusability**
- **Ease of maintenance**



Handling Variables in PL/SQL

- **Declare and initialize variables in the declaration section.**
- **Assign new values to variables in the executable section.**
- **Pass values into PL/SQL blocks through parameters.**
- **View results through output variables.**

Types of Variables

- **PL/SQL variables:**
 - Scalar
 - Composite
 - Reference
 - LOB (large objects)
- **Non-PL/SQL variables: Bind and host variables**



Using *i*SQL*Plus Variables Within PL/SQL Blocks

- **PL/SQL does not have input or output capability of its own.**
- **You can reference substitution variables within a PL/SQL block with a preceding ampersand.**
- ***i*SQL*Plus host (or “bind”) variables can be used to pass run time values out of the PL/SQL block back to the *i*SQL*Plus environment.**

Types of Variables

TRUE



256120.08

25-JAN-01

"Four score and seven years ago
our fathers brought forth upon
this continent, a new nation,
conceived in LIBERTY, and dedicated
to the proposition that all men
are created equal."



Atlanta



Declaring PL/SQL Variables

Syntax:

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

Examples:

```
DECLARE  
    v_hiredate      DATE ;  
    v_deptno        NUMBER(2) NOT NULL := 10 ;  
    v_location       VARCHAR2(13) := 'Atlanta' ;  
    c_comm           CONSTANT NUMBER := 1400 ;
```



Guidelines for Declaring PL/SQL Variables

- Follow naming conventions.
- Initialize variables designated as NOT NULL and CONSTANT.
- Declare one identifier per line.
- Initialize identifiers by using the assignment operator (`:=`) or the DEFAULT reserved word.

```
identifier := expr;
```

Naming Rules

- Two variables can have the same name, provided they are in different blocks.
- The variable name (identifier) should not be the same as the name of table columns used in the block.

```
DECLARE
    employee_id NUMBER(6);
BEGIN
    SELECT      employee_id
    INTO        employee_id
    FROM        employees
    WHERE       last_name = 'Kochhar';
END;
/
```

**Adopt a naming convention for PL/SQL identifiers:
for example,
v_employee_id**

Variable Initialization and Keywords

- Assignment operator (:=)
- DEFAULT keyword
- NOT NULL constraint

Syntax:

```
identifier := expr;
```

Examples:

```
v_hiredate := '01-JAN-2001';
```

```
v_ename := 'Maduro';
```



Scalar Data Types

- Hold a single value
- Have no internal components

25-OCT-99

256120.08

"Four score and seven years ago our fathers brought forth upon this continent, a new nation, conceived in LIBERTY, and dedicated to the proposition that all men are created equal."

TRUE

Atlanta

A yellow speech bubble contains a portion of the Gettysburg Address. The word "TRUE" is overlaid in large blue letters. The word "Atlanta" is written in pink at the bottom right of the bubble.

Base Scalar Data Types

- **CHAR [(*maximum_length*)]**
- **VARCHAR2 (*maximum_length*)**
- **LONG**
- **LONG RAW**
- **NUMBER [(*precision, scale*)]**
- **BINARY_INTEGER**
- **PLS_INTEGER**
- **BOOLEAN**

Base Scalar Data Types

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE
- TIMESTAMP WITH LOCAL TIME ZONE
- INTERVAL YEAR TO MONTH
- INTERVAL DAY TO SECOND



Scalar Variable Declarations

Examples:

```
DECLARE
    v_job          VARCHAR2(9);
    v_count        BINARY_INTEGER := 0;
    v_total_sal   NUMBER(9,2)  := 0;
    v_orderdate   DATE := SYSDATE + 7;
    c_tax_rate    CONSTANT NUMBER(3,2) := 8.25;
    v_valid        BOOLEAN NOT NULL := TRUE;
    ...

```

The %TYPE Attribute

- **Declare a variable according to:**
 - A database column definition
 - Another previously declared variable
- **Prefix %TYPE with:**
 - The database table and column
 - The previously declared variable name

Declaring Variables with the %TYPE Attribute

Syntax:

```
identifier      Table.column_name%TYPE;
```

Examples:

```
...
  v_name          employees.last_name%TYPE;
  v_balance       NUMBER(7,2);
  v_min_balance  v_balance%TYPE := 10;
...

```

Declaring Boolean Variables

- Only the values TRUE, FALSE, and NULL can be assigned to a Boolean variable.
- The variables are compared by the logical operators AND, OR, and NOT.
- The variables always yield TRUE, FALSE, or NULL.
- Arithmetic, character, and date expressions can be used to return a Boolean value.

Composite Data Types

TRUE	23-DEC-98	ATLANTA	
-------------	------------------	----------------	---

PL/SQL table structure

1	SMITH
2	JONES
3	NANCY
4	TIM

PL/SQL table structure

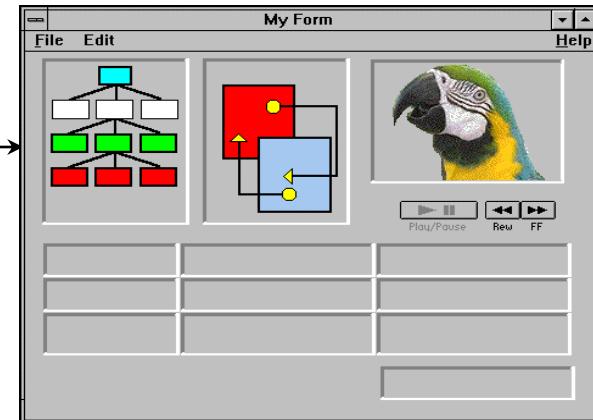
1	5000
2	2345
3	12
4	3456

VARCHAR2
BINARY_INTEGER

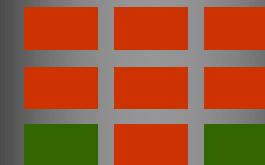
NUMBER
BINARY_INTEGER

Bind Variables

O/S
Bind variable



Server



Using Bind Variables

To reference a bind variable in PL/SQL, you must prefix its name with a colon (:).

Example:

```
VARIABLE      g_salary NUMBER
BEGIN
    SELECT      salary
    INTO        :g_salary
    FROM        employees
    WHERE       employee_id = 178;
END ;
/
PRINT g_salary
```



Referencing Non-PL/SQL Variables

Store the annual salary into a *i*SQL*Plus host variable.

```
:g_monthly_sal := v_sal / 12;
```

- Reference non-PL/SQL variables as host variables.
- Prefix the references with a colon (:).



DBMS_OUTPUT.PUT_LINE

- An Oracle-supplied packaged procedure
- An alternative for displaying data from a PL/SQL block
- Must be enabled in iSQL*Plus with
SET SERVEROUTPUT ON

```
SET SERVEROUTPUT ON
DEFINE p_annual_sal = 60000
```

```
DECLARE
    v_sal NUMBER(9,2) := &p_annual_sal;
BEGIN
    v_sal := v_sal/12;
    DBMS_OUTPUT.PUT_LINE ('The monthly salary is ' ||
                           TO_CHAR(v_sal));
END;
/
```

Summary

In this lesson you should have learned that:

- **PL/SQL blocks are composed of the following sections:**
 - Declarative (optional)
 - Executable (required)
 - Exception handling (optional)
- **A PL/SQL block can be an anonymous block, procedure, or function.**





Writing Executable Statements





Objectives

After completing this lesson, you should be able to do the following:

- **Describe the significance of the executable section**
- **Use identifiers correctly**
- **Write statements in the executable section**
- **Describe the rules of nested blocks**
- **Execute and test a PL/SQL block**
- **Use coding conventions**



PL/SQL Block Syntax and Guidelines

- **Statements can continue over several lines.**
- **Lexical units can be classified as:**
 - Delimiters
 - Identifiers
 - Literals
 - Comments

Identifiers

- **Can contain up to 30 characters**
- **Must begin with an alphabetic character**
- **Can contain numerals, dollar signs, underscores, and number signs**
- **Cannot contain characters such as hyphens, slashes, and spaces**
- **Should not have the same name as a database table column name**
- **Should not be reserved words**

PL/SQL Block Syntax and Guidelines

● Literals

- Character and date literals must be enclosed in single quotation marks.

```
v_name := 'Henderson';
```

- Numbers can be simple values or scientific notation.
- A slash (/) runs the PL/SQL block in a script file or in some tools such as *iSQL*PLUS*.

Commenting Code

- **Prefix single-line comments with two dashes (--) .**
- **Place multiple-line comments between the symbols /* and */ .**

```
DECLARE
  ...
  v_sal NUMBER (9,2);
BEGIN
  /* Compute the annual salary based on the
     monthly salary input from the user */
  v_sal := :g_monthly_sal * 12;
END;      -- This is the end of the block
```

SQL Functions in PL/SQL

- **Available in procedural statements:**

- Single-row number
- Single-row character
- Data type conversion
- Date
- Timestamp
- GREATEST and LEAST
- Miscellaneous functions



Same as in SQL

- **Not available in procedural statements:**

- DECODE
- Group functions



SQL Functions in PL/SQL: Examples

- Build the mailing list for a company.

```
v_mailing_address := v_name || CHR(10) ||
                     v_address || CHR(10) || v_state ||
                     CHR(10) || v_zip;
```

- Convert the employee name to lowercase.

```
v_ename          := LOWER(v_ename);
```



Data Type Conversion

- Convert data to comparable data types.
- Mixed data types can result in an error and affect performance.
- Conversion functions:
 - TO_CHAR
 - TO_DATE
 - TO_NUMBER

```
DECLARE
    v_date DATE := TO_DATE('12-JAN-2001', 'DD-MON-YYYY');
BEGIN
    . . .
END;
```



Data Type Conversion

This statement produces a compilation error if the variable `v_date` is declared as a DATE data type.

```
v_date := 'January 13, 2001';
```



Data Type Conversion

To correct the error, use the `TO_DATE` conversion function.

```
v_date := TO_DATE ('January 13, 2001',  
                     'Month DD, YYYY');
```

Nested Blocks and Variable Scope

- PL/SQL blocks can be nested wherever an executable statement is allowed.
- A nested block becomes a statement.
- An exception section can contain nested blocks.
- The scope of an identifier is that region of a program unit (block, subprogram, or package) from which you can reference the identifier.

Nested Blocks and Variable Scope

Example:

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

Scope of x

Scope of y



Identifier Scope

An identifier is visible in the regions where you can reference the identifier without having to qualify it:

- A block can look up to the enclosing block.
- A block cannot look down to enclosed blocks.

Qualify an Identifier

- The qualifier can be the label of an enclosing block.
- Qualify an identifier by using the block label prefix.

```
<<outer>>
  DECLARE
    birthdate DATE;
BEGIN
  DECLARE
    birthdate DATE;
BEGIN
  ...
  outer.birthdate :=
    TO_DATE('03-AUG-1976',
            'DD-MON-YYYY');
END;
...
END;
```

Determining Variable Scope

Class Exercise

```
<<outer>>

DECLARE
    v_sal      NUMBER(7,2) := 60000;
    v_comm     NUMBER(7,2) := v_sal * 0.20;
    v_message  VARCHAR2(255) := ' eligible for commission';

BEGIN
    DECLARE
        v_sal          NUMBER(7,2) := 50000;
        v_comm         NUMBER(7,2) := 0;
        v_total_comp   NUMBER(7,2) := v_sal + v_comm;
    BEGIN
        v_message := 'CLERK not' || v_message;
        outer.v_comm := v_sal * 0.30;
    END;
    v_message := 'SALESMAN' || v_message;
END;
```

1 → END;
2 → END;

Operators in PL/SQL

- Logical
 - Arithmetic
 - Concatenation
 - Parentheses to control order of operations
- 
- Same as in SQL
-
- Exponential operator (**)

Operators in PL/SQL

Examples:

- Increment the counter for a loop.

```
v_count      := v_count + 1;
```

- Set the value of a Boolean flag.

```
v_equal      := (v_n1 = v_n2);
```

- Validate whether an employee number contains a value.

```
v_valid      := (v_empno IS NOT NULL);
```



Programming Guidelines

Make code maintenance easier by:

- **Documenting code with comments**
- **Developing a case convention for the code**
- **Developing naming conventions for identifiers and other objects**
- **Enhancing readability by indenting**



Indenting Code

For clarity, indent each level of code.

Example:

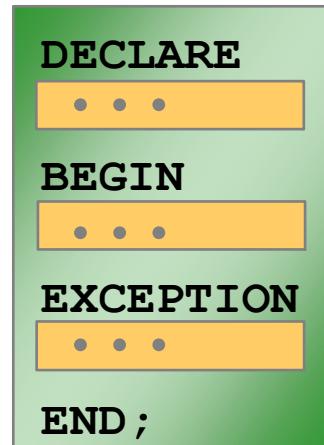
```
BEGIN
    IF x=0 THEN
        y:=1;
    END IF;
END;
```

```
DECLARE
    v_deptno          NUMBER(4);
    v_location_id    NUMBER(4);
BEGIN
    SELECT department_id,
           location_id
      INTO v_deptno,
            v_location_id
     FROM departments
    WHERE department_name
          = 'Sales';
    ...
END;
/
```

Summary

In this lesson you should have learned that:

- **PL/SQL block syntax and guidelines**
- **How to use identifiers correctly**
- **PL/SQL block structure: nesting blocks and scoping rules**
- **PL/SQL programming:**
 - Functions
 - Data type conversions
 - Operators
 - Conventions and guidelines





Interacting with the Oracle Server





Objectives

After completing this lesson, you should be able to do the following:

- Write a successful SELECT statement in PL/SQL
- Write DML statements in PL/SQL
- Control transactions in PL/SQL
- Determine the outcome of SQL data manipulation language (DML) statements



SQL Statements in PL/SQL

- Extract a row of data from the database by using the SELECT command.
- Make changes to rows in the database by using DML commands.
- Control a transaction with the COMMIT, ROLLBACK, or SAVEPOINT command.
- Determine DML outcome with implicit cursor attributes.



SELECT Statements in PL/SQL

Retrieve data from the database with a SELECT statement.

Syntax:

```
SELECT  select_list
INTO    {variable_name[, variable_name] ...
        | record_name}
FROM    table
[WHERE  condition];
```

SELECT Statements in PL/SQL

- The **INTO clause is required.**
- Queries must return one and only one row.

Example:

```
DECLARE
    v_deptno          NUMBER(4);
    v_location_id     NUMBER(4);
BEGIN
    SELECT      department_id, location_id
    INTO        v_deptno, v_location_id
    FROM        departments
    WHERE       department_name = 'Sales';
    ...
END;
/
```



Retrieving Data in PL/SQL

Retrieve the hire date and the salary for the specified employee.

Example:

```
DECLARE
    v_hire_date    employees.hire_date%TYPE;
    v_salary        employees.salary%TYPE;
BEGIN
    SELECT    hire_date, salary
    INTO      v_hire_date, v_salary
    FROM      employees
    WHERE     employee_id = 100;
    ...
END;
/
```



Retrieving Data in PL/SQL

Return the sum of the salaries for all employees in the specified department.

Example:

```
SET SERVEROUTPUT ON
DECLARE
    v_sum_sal    NUMBER(10,2);
    v_deptno     NUMBER NOT NULL := 60;
BEGIN
    SELECT      SUM(salary)  -- group function
    INTO        v_sum_sal
    FROM       employees
    WHERE      department_id = v_deptno;
    DBMS_OUTPUT.PUT_LINE ('The sum salary is ' ||
                           TO_CHAR(v_sum_sal));
END;
/
```



Naming Conventions

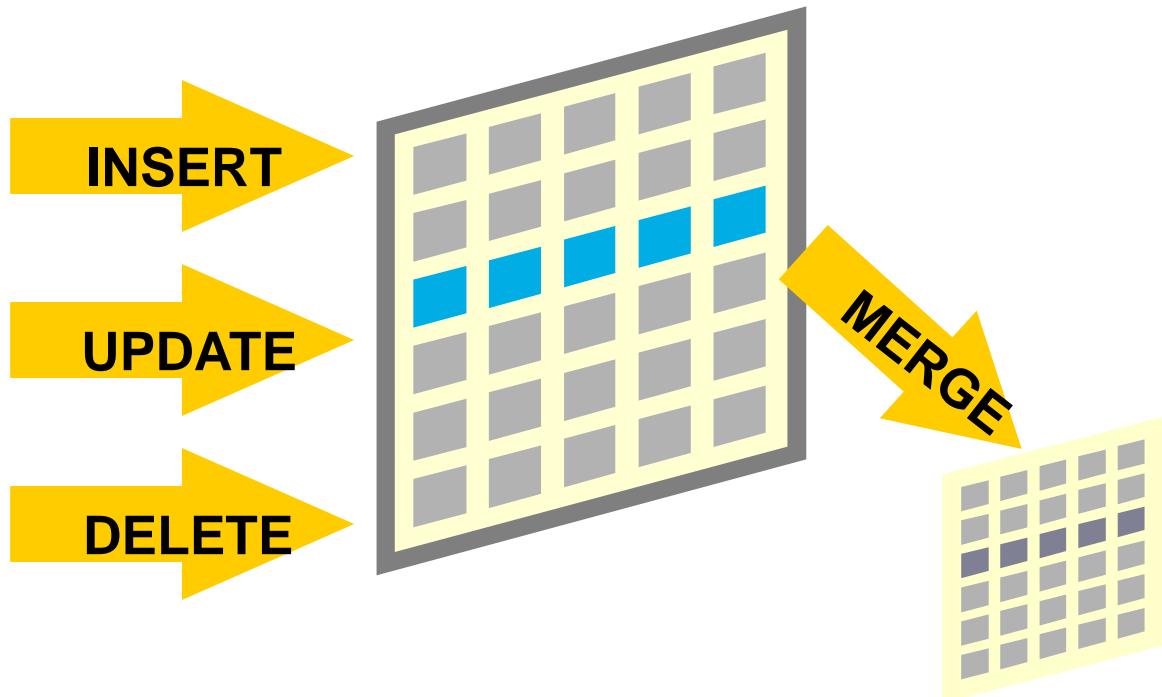
```
DECLARE
    hire_date      employees.hire_date%TYPE;
    sysdate        hire_date%TYPE;
    employee_id   employees.employee_id%TYPE := 176;
BEGIN
    SELECT      hire_date, sysdate
    INTO        hire_date, sysdate
    FROM        employees
    WHERE       employee_id = employee_id;
END ;
/
```

```
DECLARE
*
ERROR at line 1:
ORA-01422: exact fetch returns more than requested number of rows
ORA-06512: at line 6
```

Manipulating Data Using PL/SQL

Make changes to database tables by using DML commands:

- **INSERT**
- **UPDATE**
- **DELETE**
- **MERGE**





Inserting Data

Add new employee information to the EMPLOYEES table.

Example:

```
BEGIN
    INSERT INTO employees
    (employee_id, first_name, last_name, email,
     hire_date, job_id, salary)
    VALUES
    (employees_seq.NEXTVAL, 'Ruth', 'Cores', 'RCORES',
     sysdate, 'AD_ASST', 4000);
END;
/
```



Updating Data

Increase the salary of all employees who are stock clerks.

Example:

```
DECLARE
    v_sal_increase    employees.salary%TYPE := 800;
BEGIN
    UPDATE      employees
    SET          salary = salary + v_sal_increas
    WHERE        job_id = 'ST_CLERK';
END ;
/
```



Deleting Data

Delete rows that belong to department 10 from the EMPLOYEES table.

Example:

```
DECLARE
    v_deptno    employees.department_id%TYPE := 10;
BEGIN
    DELETE FROM    employees
    WHERE          department_id = v_deptno;
END;
/
```



Merging Rows

Insert or update rows in the COPY_EMP table to match the EMPLOYEES table.

```
DECLARE
    v_empno employees.employee_id%TYPE := 100;
BEGIN
MERGE INTO copy_emp c
    USING employees e
    ON (e.employee_id = v_empno)
WHEN MATCHED THEN
    UPDATE SET
        c.first_name      = e.first_name,
        c.last_name       = e.last_name,
        c.email           = e.email,
        . . .
WHEN NOT MATCHED THEN
    INSERT VALUES(e.employee_id, e.first_name, e.last_name,
        . . . , e.department_id);
END;
```



Naming Conventions

- Use a naming convention to avoid ambiguity in the WHERE clause.
- Database columns and identifiers should have distinct names.
- Syntax errors can arise because PL/SQL checks the database first for a column in the table.
- The names of local variables and formal parameters take precedence over the names of database tables.
- The names of database table columns take precedence over the names of local variables.

SQL Cursor

- A cursor is a private SQL work area.
- There are two types of cursors:
 - Implicit cursors
 - Explicit cursors
- The Oracle server uses implicit cursors to parse and execute your SQL statements.
- Explicit cursors are explicitly declared by the programmer.



SQL Cursor Attributes

Using SQL cursor attributes, you can test the outcome of your SQL statements.

SQL%ROWCOUNT	Number of rows affected by the most recent SQL statement (an integer value)
SQL%FOUND	Boolean attribute that evaluates to TRUE if the most recent SQL statement affects one or more rows
SQL%NOTFOUND	Boolean attribute that evaluates to TRUE if the most recent SQL statement does not affect any rows
SQL%ISOPEN	Always evaluates to FALSE because PL/SQL closes implicit cursors immediately after they are executed



SQL Cursor Attributes

Delete rows that have the specified employee ID from the EMPLOYEES table. Print the number of rows deleted.

Example:

```
VARIABLE rows_deleted VARCHAR2(30)
DECLARE
    v_employee_id employees.employee_id%TYPE := 176;
BEGIN
    DELETE FROM employees
    WHERE employee_id = v_employee_id;
    :rows_deleted := (SQL%ROWCOUNT ||
                      ' row deleted.');
END;
/
PRINT rows_deleted
```



Transaction Control Statements

- **Initiate a transaction with the first DML command to follow a COMMIT or ROLLBACK.**
- **Use COMMIT and ROLLBACK SQL statements to terminate a transaction explicitly.**



Summary

In this lesson you should have learned how to:

- **Embed SQL in the PL/SQL block using SELECT, INSERT, UPDATE, DELETE, and MERGE**
- **Embed transaction control statements in a PL/SQL block COMMIT, ROLLBACK, and SAVEPOINT**

Summary

In this lesson you should have learned that:

- **There are two cursor types: implicit and explicit.**
- **Implicit cursor attributes are used to verify the outcome of DML statements:**
 - SQL%ROWCOUNT
 - SQL%FOUND
 - SQL%NOTFOUND
 - SQL%ISOPEN
- **Explicit cursors are defined by the programmer.**



Writing Control Structures

www.



.com





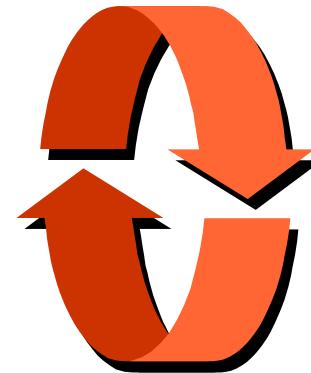
Objectives

After completing this lesson, you should be able to do the following:

- **Identify the uses and types of control structures**
- **Construct an IF statement**
- **Use CASE expressions**
- **Construct and identify different loop statements**
- **Use logic tables**
- **Control block flow using nested loops and labels**

Controlling PL/SQL Flow of Execution

- You can change the logical execution of statements using conditional IF statements and loop control structures.
- Conditional IF statements:
 - IF-THEN-END IF
 - IF-THEN-ELSE-END IF
 - IF-THEN-ELSIF-END IF





IF Statements

Syntax:

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

If the employee name is Gietz, set the Manager ID to 102.

```
IF UPPER(v_last_name) = 'GIETZ' THEN  
    v_mgr := 102;  
END IF;
```



Simple IF Statements

If the last name is Vargas:

- Set job ID to SA_REP
- Set department number to 80

```
.
.
.
IF v_ename      = 'Vargas' THEN
    v_job        := 'SA_REP';
    v_deptno     := 80;
END IF;
.
.
```



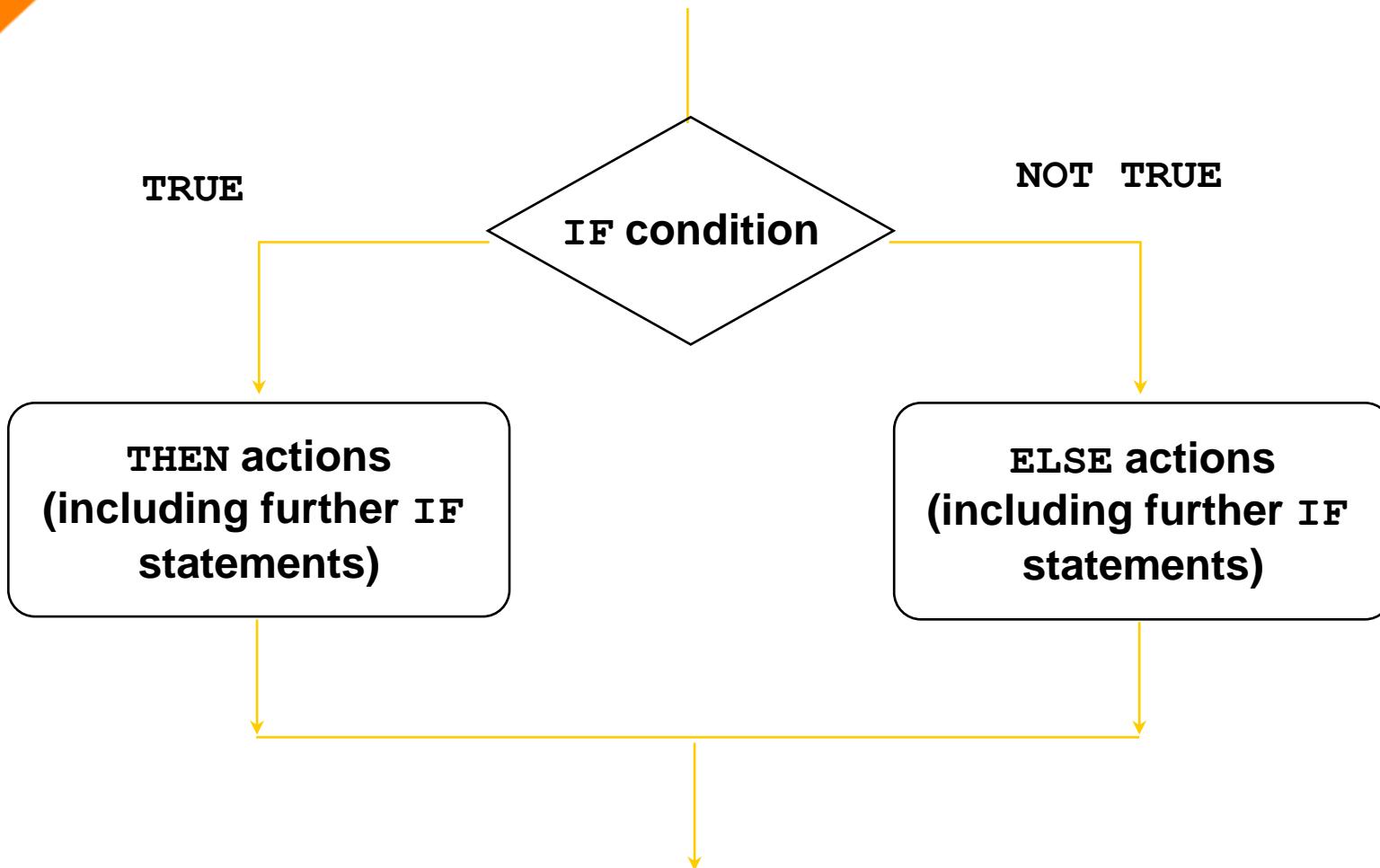
Compound IF Statements

If the last name is Vargas and the salary is more than 6500:

Set department number to 60.

```
. . .
IF v_ename = 'Vargas' AND salary > 6500 THEN
    v_deptno := 60;
END IF;
. . .
```

IF-THEN-ELSE Statement Execution Flow



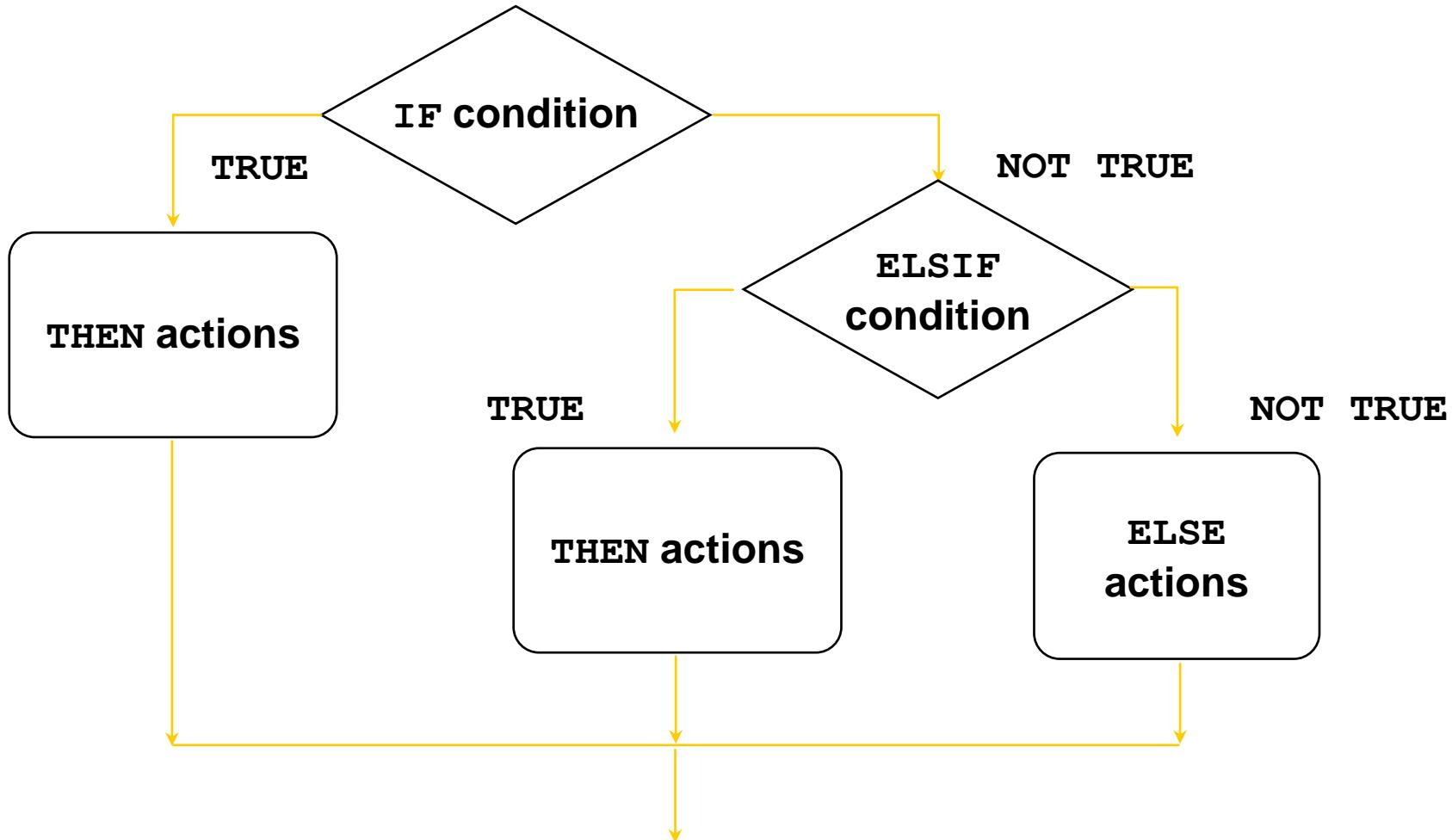


IF-THEN-ELSE Statements

Set a Boolean flag to TRUE if the hire date is greater than five years; otherwise, set the Boolean flag to FALSE.

```
DECLARE
    v_hire_date DATE := '12-Dec-1990';
    v_five_years BOOLEAN;
BEGIN
    .
    .
    .
    IF MONTHS_BETWEEN(SYSDATE,v_hire_date)/12 > 5 THEN
        v_five_years := TRUE;
    ELSE
        v_five_years := FALSE;
    END IF;
    .
    .
```

IF-THEN-ELSIF Statement Execution Flow





IF-THEN-ELSIF Statements

For a given value, calculate a percentage of that value based on a condition.

Example:

```
. . .
IF      v_start > 100 THEN
    v_start := 0.2 * v_start;
ELSIF  v_start >= 50 THEN
    v_start := 0.5 * v_start;
ELSE
    v_start := 0.1 * v_start;
END IF;
. . .
```



CASE Expressions

- A CASE expression selects a result and returns it.
- To select the result, the CASE expression uses an expression whose value is used to select one of several alternatives.

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



CASE Expressions: Example

```
SET SERVEROUTPUT ON
DECLARE
    v_grade CHAR(1) := UPPER('&p_grade');
    v_appraisal VARCHAR2(20);
BEGIN
    v_appraisal :=
        CASE v_grade
            WHEN 'A' THEN 'Excellent'
            WHEN 'B' THEN 'Very Good'
            WHEN 'C' THEN 'Good'
            ELSE 'No such grade'
        END;
    DBMS_OUTPUT.PUT_LINE ('Grade: '|| v_grade || '
                           Appraisal ' || v_appraisal);
END;
/
```



Handling Nulls

When working with nulls, you can avoid some common mistakes by keeping in mind the following rules:

- Simple comparisons involving nulls always yield NULL.**
- Applying the logical operator NOT to a null yields NULL.**
- In conditional control statements, if the condition yields NULL, its associated sequence of statements is not executed.**

Logic Tables

Build a simple Boolean condition with a comparison operator.

AND	<i>TRUE</i>	<i>FALSE</i>	<i>NULL</i>
<i>TRUE</i>	TRUE	FALSE	NULL
<i>FALSE</i>	FALSE	FALSE	FALSE
<i>NULL</i>	NULL	FALSE	NULL

OR	<i>TRUE</i>	<i>FALSE</i>	<i>NULL</i>
<i>TRUE</i>	TRUE	TRUE	TRUE
<i>FALSE</i>	TRUE	FALSE	NULL
<i>NULL</i>	TRUE	NULL	NULL

NOT	
<i>TRUE</i>	FALSE
<i>FALSE</i>	TRUE
<i>NULL</i>	NULL

Boolean Conditions

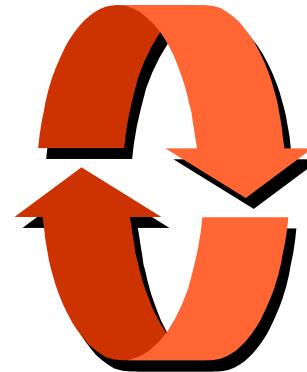
What is the value of v_flag in each case?

```
v_flag := v_reordered_flag AND v_available_flag;
```

V_reordered_flag	V_available_flag	V_flag
TRUE	TRUE	?
TRUE	FALSE	?
NULL	TRUE	?
NULL	FALSE	?

Iterative Control: LOOP Statements

- Loops repeat a statement or sequence of statements multiple times.
- There are three loop types:
 - Basic loop
 - FOR loop
 - WHILE loop





Basic Loops

Syntax:

```
LOOP                                -- delimiter  
    statement1;  
    . . .  
    EXIT [WHEN condition];          -- EXIT statement  
END LOOP;                            -- delimiter
```

condition is a Boolean variable or
 expression (TRUE, FALSE, or NULL);



Basic Loops

Example:

```
DECLARE
    v_country_id      locations.country_id%TYPE := 'CA';
    v_location_id     locations.location_id%TYPE;
    v_counter          NUMBER(2) := 1;
    v_city             locations.city%TYPE := 'Montreal';
BEGIN
    SELECT MAX(location_id) INTO v_location_id FROM locations
    WHERE country_id = v_country_id;
LOOP
    INSERT INTO locations(location_id, city, country_id)
    VALUES((v_location_id + v_counter),v_city, v_country_id);
    v_counter := v_counter + 1;
    EXIT WHEN v_counter > 3;
END LOOP;
END;
/
```



WHILE Loops

Syntax:

```
WHILE condition LOOP ←  
      statement1;  
      statement2;  
      . . .  
END LOOP;
```

Condition is evaluated at the beginning of each iteration.

Use the WHILE loop to repeat statements while a condition is TRUE.



WHILE Loops

Example:

```
DECLARE
    v_country_id      locations.country_id%TYPE := 'CA';
    v_location_id     locations.location_id%TYPE;
    v_city             locations.city%TYPE := 'Montreal';
    v_counter          NUMBER   := 1;

BEGIN
    SELECT MAX(location_id) INTO v_location_id FROM locations
    WHERE country_id = v_country_id;
    WHILE v_counter <= 3 LOOP
        INSERT INTO locations(location_id, city, country_id)
        VALUES((v_location_id + v_counter), v_city, v_country_id);
        v_counter := v_counter + 1;
    END LOOP;
END ;
/
```



FOR Loops

Syntax:

```
FOR counter IN [REVERSE]
    lower_bound..upper_bound LOOP
    statement1;
    statement2;
    .
    .
    END LOOP;
```

- Use a FOR loop to shortcut the test for the number of iterations.
- Do not declare the counter; it is declared implicitly.
- 'lower_bound .. upper_bound' is required syntax.



FOR Loops

Insert three new locations IDs for the country code of CA and the city of Montreal.

```
DECLARE
    v_country_id      locations.country_id%TYPE := 'CA';
    v_location_id     locations.location_id%TYPE;
    v_city            locations.city%TYPE := 'Montreal';
BEGIN
    SELECT MAX(location_id) INTO v_location_id
        FROM locations
        WHERE country_id = v_country_id;
    FOR i IN 1..3 LOOP
        INSERT INTO locations(location_id, city, country_id)
        VALUES((v_location_id + i), v_city, v_country_id );
    END LOOP;
END;
/
```

FOR Loops

Guidelines

- **Reference the counter within the loop only; it is undefined outside the loop.**
- **Do *not* reference the counter as the target of an assignment.**



Guidelines While Using Loops

- Use the **basic loop** when the statements inside the loop must execute at least once.
- Use the **WHILE loop** if the condition has to be evaluated at the start of each iteration.
- Use a **FOR loop** if the number of iterations is known.



Nested Loops and Labels

- Nest loops to multiple levels.
- Use labels to distinguish between blocks and loops.
- Exit the outer loop with the `EXIT` statement that references the label.



Nested Loops and Labels

```
...
BEGIN
  <<Outer_loop>>
  LOOP
    v_counter := v_counter+1;
  EXIT WHEN v_counter>10;
  <<Inner_loop>>
  LOOP
    ...
    EXIT Outer_loop WHEN total_done = 'YES';
    -- Leave both loops
    EXIT WHEN inner_done = 'YES';
    -- Leave inner loop only
    ...
  END LOOP Inner_loop;
  ...
END LOOP Outer_loop;
END;
```



Summary

In this lesson you should have learned to:
Change the logical flow of statements by using control structures.

- **Conditional (IF statement)**
- **CASE Expressions**
- **Loops:**
 - Basic loop
 - FOR loop
 - WHILE loop
- **EXIT statements**



Working with Composite Data Types





Objectives

After completing this lesson, you should be able to do the following:

- **Create user-defined PL/SQL records**
- **Create a record with the %ROWTYPE attribute**
- **Create an INDEX BY table**
- **Create an INDEX BY table of records**
- **Describe the difference between records, tables, and tables of records**

Composite Data Types

- **Are of two types:**
 - PL/SQL RECORDS
 - PL/SQL Collections
 - INDEX BY Table
 - Nested Table
 - VARRAY
- **Contain internal components**
- **Are reusable**



PL/SQL Records

- Must contain one or more components of any scalar, RECORD, or INDEX BY table data type, called fields
- Are similar in structure to records in a third generation language (3GL)
- Are not the same as rows in a database table
- Treat a collection of fields as a logical unit
- Are convenient for fetching a row of data from a table for processing



Creating a PL/SQL Record

Syntax:

```
TYPE type_name IS RECORD  
    (field_declaration[, field_declaration]...);  
identifier      type_name;
```

Where *field_declaration* is:

```
field_name {field_type | variable%TYPE  
| table.column%TYPE | table%ROWTYPE}  
[ [NOT NULL] { := | DEFAULT} expr]
```



Creating a PL/SQL Record

Declare variables to store the name, job, and salary of a new employee.

Example:

```
...
TYPE emp_record_type IS RECORD
    (last_name    VARCHAR2(25),
     job_id       VARCHAR2(10),
     salary        NUMBER(8,2));
emp_record      emp_record_type;
...
```

PL/SQL Record Structure

Field1 (data type) Field2 (data type) Field3 (data type)



Example:

Field1 (data type) Field2 (data type) Field3 (data type)

employee_id number(6) last_name varchar2(25) job_id varchar2(10)



100

King

AD_PRES

The %ROWTYPE Attribute

- Declare a variable according to a collection of columns in a database table or view.
- Prefix %ROWTYPE with the database table.
- Fields in the record take their names and data types from the columns of the table or view.



Advantages of Using %ROWTYPE

- The number and data types of the underlying database columns need not be known.
- The number and data types of the underlying database column may change at run time.
- The attribute is useful when retrieving a row with the SELECT * statement.



The %ROWTYPE Attribute

Examples:

Declare a variable to store the information about a department from the DEPARTMENTS table.

```
dept_record    departments%ROWTYPE;
```

Declare a variable to store the information about an employee from the EMPLOYEES table.

```
emp_record    employees%ROWTYPE;
```



INDEX BY Tables

- **Are composed of two components:**
 - Primary key of data type BINARY_INTEGER
 - Column of scalar or record data type
- **Can increase in size dynamically because they are unconstrained**



Creating an INDEX BY Table

Syntax:

```
TYPE type_name IS TABLE OF
    {column_type | variable%TYPE
     | table.column%TYPE} [NOT NULL]
    | table.%ROWTYPE
    [INDEX BY BINARY_INTEGER];
identifier      type_name;
```

Declare an INDEX BY table to store names.

Example:

```
...
TYPE ename_table_type IS TABLE OF
    employees.last_name%TYPE
    INDEX BY BINARY_INTEGER;
ename_table ename_table_type;
...
```

INDEX BY Table Structure

Unique identifier

...
1
2
3
...

BINARY_INTEGER

Column

...
Jones
Smith
Maduro
...

Scalar

Creating an INDEX BY Table

```
DECLARE
    TYPE ename_table_type IS TABLE OF
        employees.last_name%TYPE
        INDEX BY BINARY_INTEGER;
    TYPE hiredate_table_type IS TABLE OF DATE
        INDEX BY BINARY_INTEGER;
    ename_table          ename_table_type;
    hiredate_table       hiredate_table_type;
BEGIN
    ename_table(1)      := 'CAMERON';
    hiredate_table(8)   := SYSDATE + 7;
    IF ename_table.EXISTS(1) THEN
        INSERT INTO ...
        ...
    END;
/

```

Using INDEX BY Table Methods

The following methods make INDEX BY tables easier to use:

- **EXISTS**
 - **COUNT**
 - **FIRST and LAST**
 - **PRIOR**
- **NEXT**
 - **TRIM**
 - **DELETE**



INDEX BY Table of Records

- Define a **TABLE** variable with a permitted PL/SQL data type.
- Declare a PL/SQL variable to hold department information.

Example:

```
DECLARE
  TYPE dept_table_type IS TABLE OF
    departments%ROWTYPE
    INDEX BY BINARY_INTEGER;
  dept_table dept_table_type;
  -- Each element of dept_table is a record
```



Example of INDEX BY Table of Records

```
SET SERVEROUTPUT ON
DECLARE
    TYPE emp_table_type is table of
        employees%ROWTYPE INDEX BY BINARY_INTEGER;
    my_emp_table  emp_table_type;
    v_count        NUMBER(3) := 104;
BEGIN
    FOR i IN 100..v_count
    LOOP
        SELECT * INTO my_emp_table(i) FROM employees
        WHERE employee_id = i;
    END LOOP;
    FOR i IN my_emp_table.FIRST..my_emp_table.LAST
    LOOP
        DBMS_OUTPUT.PUT_LINE(my_emp_table(i).last_name);
    END LOOP;
END;
```



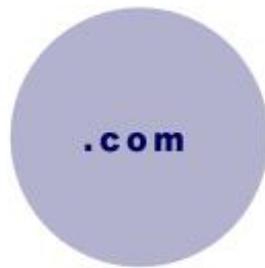
Summary

In this lesson, you should have learned to:

- **Define and reference PL/SQL variables of composite data types:**
 - PL/SQL records
 - INDEX BY tables
 - INDEX BY table of records
- **Define a PL/SQL record by using the %ROWTYPE attribute**



Writing Explicit Cursors





Objectives

After completing this lesson, you should be able to do the following:

- **Distinguish between an implicit and an explicit cursor**
- **Discuss when and why to use an explicit cursor**
- **Use a PL/SQL record variable**
- **Write a cursor FOR loop**



About Cursors

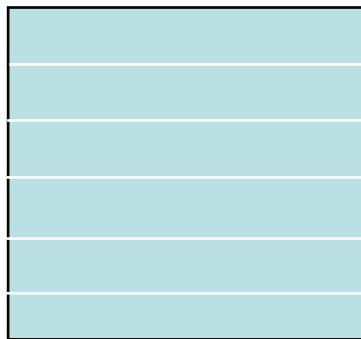
Every SQL statement executed by the Oracle Server has an individual cursor associated with it:

- **Implicit cursors: Declared for all DML and PL/SQL SELECT statements**
- **Explicit cursors: Declared and named by the programmer**

Explicit Cursor Functions

Cursor

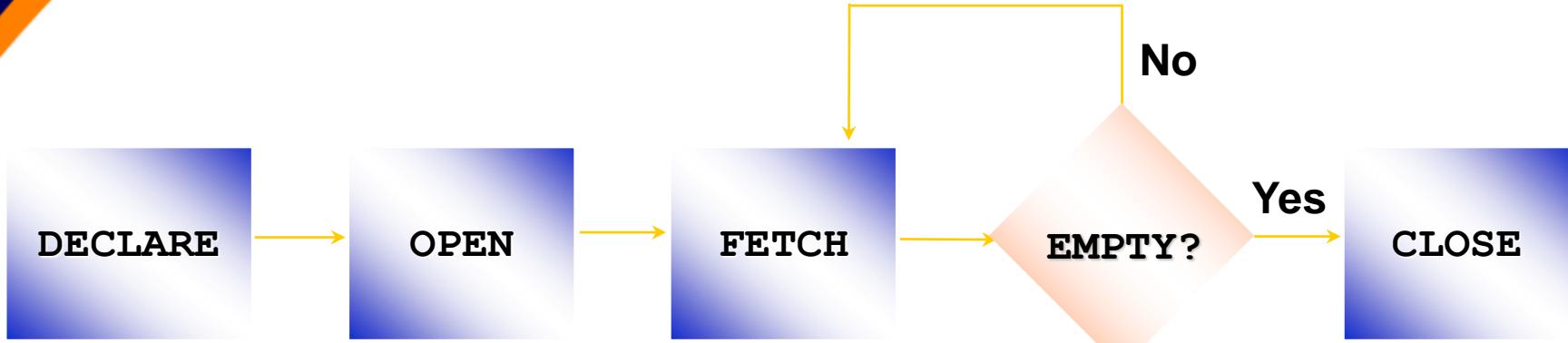
Active set



Table

100	King	AD_PRES
101	Kochhar	AD_VP
102	De Haan	AD_VP
.	.	.
.	.	.
.	.	.
139	Seo	ST_CLERK
140	Patel	ST_CLERK
.	.	.

Controlling Explicit Cursors

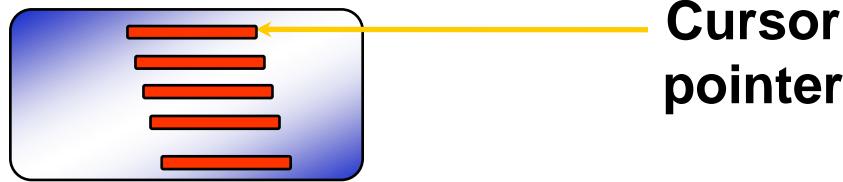


- Create a named SQL area
- Identify the active set
- Load the current row into variables
- Test for existing rows
- Return to **FETCH** if rows are found
- Release the active set

Controlling Explicit Cursors

1. Open the cursor
2. Fetch a row
3. Close the Cursor

1. Open the cursor.

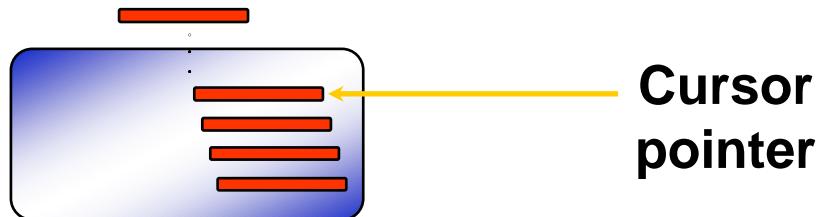


Cursor
pointer

Controlling Explicit Cursors

1. Open the cursor
2. Fetch a row
3. Close the Cursor

2. Fetch a row using the cursor.

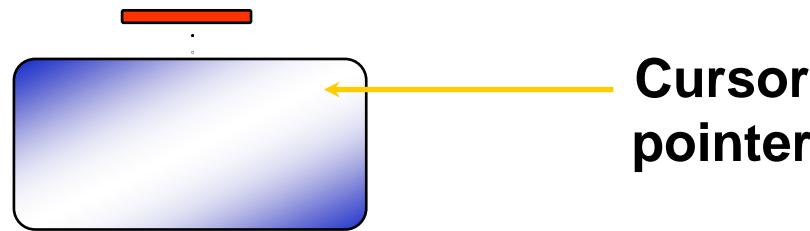


Continue until empty.

Controlling Explicit Cursors

1. Open the cursor
2. Fetch a row
3. Close the Cursor

3. Close the cursor.





Declaring the Cursor

Syntax:

```
CURSOR cursor_name IS  
    select_statement;
```

- **Do not include the INTO clause in the cursor declaration.**
- **If processing rows in a specific sequence is required, use the ORDER BY clause in the query.**



Declaring the Cursor

Example:

```
DECLARE
    CURSOR emp_cursor IS
        SELECT employee_id, last_name
        FROM employees;

    CURSOR dept_cursor IS
        SELECT *
        FROM departments
        WHERE location_id = 170;
BEGIN
    ...

```



Opening the Cursor

Syntax:

```
OPEN  cursor_name;
```

- **Open the cursor to execute the query and identify the active set.**
- **If the query returns no rows, no exception is raised.**
- **Use cursor attributes to test the outcome after a fetch.**



Fetching Data from the Cursor

Syntax:

```
FETCH cursor_name INTO [variable1, variable2, ...]  
| record_name];
```

- **Retrieve the current row values into variables.**
- **Include the same number of variables.**
- **Match each variable to correspond to the columns positionally.**
- **Test to see whether the cursor contains rows.**



Fetching Data from the Cursor

Example:

```
LOOP
    FETCH emp_cursor INTO v_empno,v_ename;
    EXIT WHEN ...;
    ...
    -- Process the retrieved data
    ...
END LOOP;
```



Closing the Cursor

Syntax:

```
CLOSE      cursor_name;
```

- Close the cursor after completing the processing of the rows.
- Reopen the cursor, if required.
- Do not attempt to fetch data from a cursor after it has been closed.



Explicit Cursor Attributes

Obtain status information about a cursor.

Attribute	Type	Description
%ISOPEN	Boolean	Evaluates to TRUE if the cursor is open
%NOTFOUND	Boolean	Evaluates to TRUE if the most recent fetch does not return a row
%FOUND	Boolean	Evaluates to TRUE if the most recent fetch returns a row; complement of %NOTFOUND
%ROWCOUNT	Number	Evaluates to the total number of rows returned so far



The %ISOPEN Attribute

- Fetch rows only when the cursor is open.
- Use the %ISOPEN cursor attribute before performing a fetch to test whether the cursor is open.

Example:

```
IF NOT emp_cursor%ISOPEN THEN
    OPEN emp_cursor;
END IF;
LOOP
    FETCH emp_cursor...
```



Controlling Multiple Fetches

- **Process several rows from an explicit cursor using a loop.**
- **Fetch a row with each iteration.**
- **Use explicit cursor attributes to test the success of each fetch.**



The %NOTFOUND and %ROWCOUNT Attributes

- Use the **%ROWCOUNT cursor attribute** to retrieve an exact number of rows.
- Use the **%NOTFOUND cursor attribute** to determine when to exit the loop.



Example

```
DECLARE
    v_empno employees.employee_id%TYPE;
    v_ename employees.last_name%TYPE;
CURSOR emp_cursor IS
    SELECT employee_id, last_name
    FROM employees;
BEGIN
    OPEN emp_cursor;
    LOOP
        FETCH emp_cursor INTO v_empno, v_ename;
        EXIT WHEN emp_cursor%ROWCOUNT > 10 OR
                           emp_cursor%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE (TO_CHAR(v_empno)
                               || ' ' || v_ename);
    END LOOP;
    CLOSE emp_cursor;
END ;
```



Cursors and Records

Process the rows of the active set by fetching values into a PL/SQL RECORD.

```
DECLARE
    CURSOR emp_cursor IS
        SELECT employee_id, last_name
        FROM employees;
    emp_record    emp_cursor%ROWTYPE;
BEGIN
    OPEN emp_cursor;
    LOOP
        FETCH emp_cursor INTO emp_record;
        ...
    END LOOP;
END;
```

emp_record
employee_id

last_name

100

King



Cursor FOR Loops

Syntax:

```
FOR record_name IN cursor_name LOOP  
    statement1;  
    statement2;  
    . . .  
END LOOP;
```

- The cursor FOR loop is a shortcut to process explicit cursors.
- Implicit open, fetch, exit, and close occur.
- The record is implicitly declared.



Cursor FOR Loops

Print a list of the employees who work for the sales department.

```
DECLARE
    CURSOR emp_cursor IS
        SELECT last_name, department_id
        FROM employees;
BEGIN
    FOR emp_record IN emp_cursor LOOP
        -- implicit open and implicit fetch occur
        IF emp_record.department_id = 80 THEN
            ...
        END LOOP; -- implicit close occurs
    END;
    /

```



Cursor FOR Loops Using Subqueries

No need to declare the cursor.

Example:

```
BEGIN
    FOR emp_record IN (SELECT last_name, department_id
                        FROM employees) LOOP
        -- implicit open and implicit fetch occur
        IF emp_record.department_id = 80 THEN
            ...
        END LOOP; -- implicit close occurs
    END;
```

Summary

In this lesson you should have learned to:

- **Distinguish cursor types:**
 - Implicit cursors: used for all DML statements and single-row queries
 - Explicit cursors: used for queries of zero, one, or more rows
- **Manipulate explicit cursors**
- **Evaluate the cursor status by using cursor attributes**
- **Use cursor FOR loops**



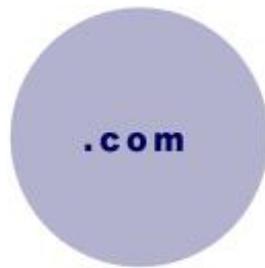
Practice 6 Overview

This practice covers the following topics:

- Declaring and using explicit cursors to query rows of a table
- Using a cursor FOR loop
- Applying cursor attributes to test the cursor status



Advanced Explicit Cursor Concepts





Objectives

After completing this lesson, you should be able to do the following:

- **Write a cursor that uses parameters**
- **Determine when a FOR UPDATE clause in a cursor is required**
- **Determine when to use the WHERE CURRENT OF clause**
- **Write a cursor that uses a subquery**

Cursors with Parameters

Syntax:

```
CURSOR cursor_name
    [ (parameter_name datatype, . . . ) ]
IS
    select_statement;
```

- Pass parameter values to a cursor when the cursor is opened and the query is executed.
- Open an explicit cursor several times with a different active set each time.

```
OPEN cursor_name(parameter_value, . . . . .) ;
```



Cursors with Parameters

Pass the department number and job title to the WHERE clause, in the cursor SELECT statement.

```
DECLARE
    CURSOR emp_cursor
    (p_deptno NUMBER, p_job VARCHAR2) IS
        SELECT employee_id, last_name
        FROM employees
        WHERE department_id = p_deptno
        AND job_id = p_job;
BEGIN
    OPEN emp_cursor (80, 'SA_REP');
    .
    .
    CLOSE emp_cursor;
    OPEN emp_cursor (60, 'IT_PROG');
    .
    .
END;
```



The FOR UPDATE Clause

Syntax:

```
SELECT ...  
FROM      ...  
FOR UPDATE [OF column_reference] [NOWAIT] ;
```

- Use explicit locking to deny access for the duration of a transaction.
- Lock the rows *before* the update or delete.



The FOR UPDATE Clause

Retrieve the employees who work in department 80 and update their salary.

```
DECLARE
    CURSOR emp_cursor IS
        SELECT employee_id, last_name, department_name
        FROM employees,departments
        WHERE employees.department_id =
              departments.department_id
        AND employees.department_id = 80
        FOR UPDATE OF salary NOWAIT;
```



The WHERE CURRENT OF Clause

Syntax:

```
WHERE CURRENT OF cursor ;
```

- Use cursors to update or delete the current row.
- Include the FOR UPDATE clause in the cursor query to lock the rows first.
- Use the WHERE CURRENT OF clause to reference the current row from an explicit cursor.



The WHERE CURRENT OF Clause

```
DECLARE
CURSOR sal_cursor IS
SELECT e.department_id, employee_id, last_name, salary
FROM employees e, departments d
WHERE d.department_id = e.department_id
    and d.department_id = 60
FOR UPDATE OF salary NOWAIT;
BEGIN
FOR emp_record IN sal_cursor
LOOP
IF emp_record.salary < 5000 THEN
    UPDATE employees
        SET salary = emp_record.salary * 1.10
        WHERE CURRENT OF sal_cursor;
END IF;
END LOOP;
END;
/
```



Cursors with Subqueries

Example:

```
DECLARE
    CURSOR my_cursor IS
        SELECT t1.department_id, t1.department_name,
               t2.staff
        FROM departments t1, (SELECT department_id,
                                         COUNT(*) AS STAFF
                                    FROM employees
                                   GROUP BY department_id) t2
       WHERE t1.department_id = t2.department_id
         AND t2.staff >= 3;
    ...

```



Summary

In this lesson, you should have learned to:

- **Return different active sets using cursors with parameters.**
- **Define cursors with subqueries and correlated subqueries.**
- **Manipulate explicit cursors with commands using the:**
 - FOR UPDATE clause
 - WHERE CURRENT OF clause



Collection Types





Collection

- A Collection is an ordered group of elements of particular data types. It can be a collection of simple data type or complex data type (like user-defined or record types).
- In the collection, each element is identified by a term called "**subscript**." Each item in the collection is assigned with a unique subscript. The data in that collection can be manipulated or fetched by referring to that unique subscript.



PL/SQL Collection Types

Collection Type	Number of Elements	Subscript Type	Dense or Sparse	Where Created	Can Be Object Type Attribute
Associative array (or index-by table)	Unbounded	String or integer	Either	Only in PL/SQL block	No
Nested table	Unbounded	Integer	Starts dense, can become sparse	Either in PL/SQL block or at schema level	Yes
Variable-size array (varray)	Bounded	Integer	Always dense	Either in PL/SQL block or at schema level	Yes



Varrays

- Varray is a collection method in which the size of the array is fixed. The array size cannot be exceeded than its fixed value.
- The subscript of the Varray is of a numeric value.

Following are the attributes of Varrays.

- Upper limit size is fixed
- Populated sequentially starting with the subscript '1'
- This collection type is always dense, i.e. we cannot delete any array elements. Varray can be deleted as a whole, or it can be trimmed from the end.



Varrays

- Since it always is dense in nature, it has very less flexibility.
- It is more appropriate to use when the array size is known and to perform similar activities on all the array elements.
- The subscript and sequence always remain stable, i.e. the subscript and count of the collection is always same.



Varrays

- They need to be initialized before using them in programs. Any operation (except EXISTS operation) on an uninitialized collection will throw an error.
- It can be created as a database object, which is visible throughout the database or inside the subprogram, which can be used only in that subprogram.



Nested Tables

- A Nested table is a collection in which the size of the array is not fixed. It has the numeric subscript type.
- The Nested table has no upper size limit.
- Since the upper size limit is not fixed, the collection, memory needs to be extended each time before we use it. We can extend the collection using 'EXTEND' keyword.
- Populated sequentially starting with the subscript '1'.



Nested Tables

- This collection type can be of both **dense and sparse**, i.e. we can create the collection as a dense, and we can also delete the individual array element randomly, which make it as sparse.
- It gives more flexibility regarding deleting the array element.
- It is stored in the system generated database table and can be used in the select query to fetch the values.



Nested Tables

- The subscript and sequence are not stable, i.e. the subscript and the count of the array element can vary.
- They need to be initialized before using them in programs. Any operation (except EXISTS operation) on the uninitialized collection will throw an error.
- It can be created as a database object, which is visible throughout the database or inside the subprogram, which can be used only in that subprogram.



Index-by-table

- Index-by-table is a collection in which the array size is not fixed. Unlike the other collection types, in the index-by-table collection the subscript can be defined by the user.
- The subscript can be integer or strings. At the time of creating the collection, the subscript type should be mentioned.
- These collections are not stored sequentially.
- They are always sparse in nature.
- The array size is not fixed.
- They cannot be stored in the database column. They shall be created and used in any program in that particular session.

Index-by-table

- They give more flexibility in terms of maintaining subscript.
- The subscripts can be of negative subscript sequence also.
- They are more appropriate to use for relatively smaller collective values in which the collection can be initialized and used within the same subprograms.
- They need not be initialized before start using them.
- It cannot be created as a database object. It can only be created inside the subprogram, which can be used only in that subprogram.
- BULK COLLECT cannot be used in this collection type as the subscript should be given explicitly for each record in the collection.



Save Point

- SAVEPOINT names and marks the current point in the processing of a transaction.
- Savepoints let you roll back part of a transaction instead of the whole transaction.
- The number of active savepoints for each session is unlimited.

We
Are IT



Handling Exceptions

www.



.com





Objectives

After completing this lesson, you should be able to do the following:

- Define PL/SQL exceptions
- Recognize unhandled exceptions
- List and use different types of PL/SQL exception handlers
- Trap unanticipated errors
- Describe the effect of exception propagation in nested blocks
- Customize PL/SQL exception messages

Handling Exceptions with PL/SQL

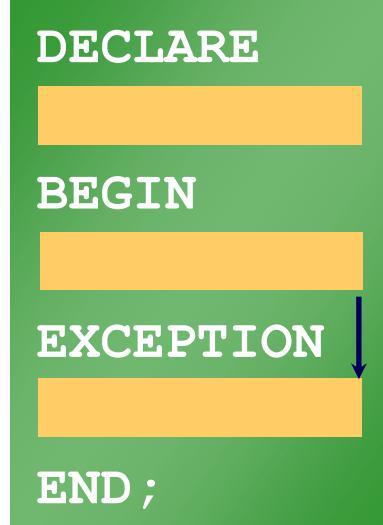
- An exception is an identifier in PL/SQL that is raised during execution.
- How is it raised?
 - An Oracle error occurs.
 - You raise it explicitly.
- How do you handle it?
 - Trap it with a handler.
 - Propagate it to the calling environment.

Handling Exceptions

Trap the exception

Exception
is raised

Exception
is trapped



Propagate the exception



Exception
is raised

Exception
is not
trapped

Exception
propagates to calling
environment

Exception Types

- Predefined Oracle Server
- Nonpredefined Oracle Server

- User-defined

} Implicitly raised



Trapping Exceptions

Syntax:

EXCEPTION

```
WHEN exception1 [OR exception2 . . .] THEN  
    statement1;  
    statement2;  
    . . .  
[WHEN exception3 [OR exception4 . . .] THEN  
    statement1;  
    statement2;  
    . . .]  
[WHEN OTHERS THEN  
    statement1;  
    statement2;  
    . . .]
```



Trapping Exceptions Guidelines

- The **EXCEPTION keyword starts exception-handling section.**
- **Several exception handlers are allowed.**
- **Only one handler is processed before leaving the block.**
- **WHEN OTHERS is the last clause.**



Trapping Predefined Oracle Server Errors

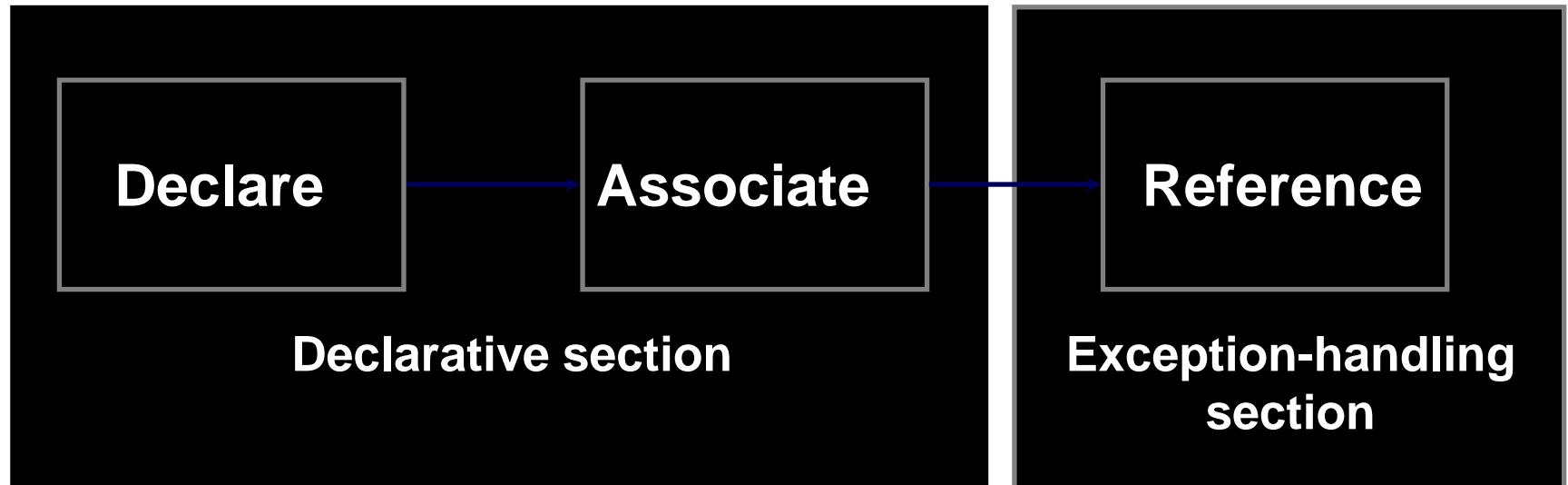
- **Reference the standard name in the exception-handling routine.**
- **Sample predefined exceptions:**
 - NO_DATA_FOUND
 - TOO_MANY_ROWS
 - INVALID_CURSOR
 - ZERO_DIVIDE
 - DUP_VAL_ON_INDEX

Predefined Exceptions

Syntax:

```
BEGIN  
  . . .  
EXCEPTION  
  WHEN NO_DATA_FOUND THEN  
    statement1;  
    statement2;  
  
  WHEN TOO_MANY_ROWS THEN  
    statement1;  
WHEN OTHERS THEN  
  statement1;  
  statement2;  
  statement3;  
END ;
```

Trapping Nonpredefined Oracle Server Errors



Name the exception

Code the PRAGMA EXCEPTION_INIT

Handle the raised exception

Nonpredefined Error

Trap for Oracle server error number –2292, an integrity constraint violation.

```
DEFINE p_deptno = 10
DECLARE
    e_emps_remaining EXCEPTION;
    PRAGMA EXCEPTION_INIT
        (e_emps_remaining, -2292);
BEGIN
    DELETE FROM departments
    WHERE department_id = &p_deptno;
    COMMIT;
EXCEPTION
    WHEN e_emps_remaining THEN
        DBMS_OUTPUT.PUT_LINE ('Cannot remove dept ' ||
            TO_CHAR(&p_deptno) || '. Employees exist. ');
END;
```

1

2

3



Functions for Trapping Exceptions

- **SQLCODE:** Returns the numeric value for the error code
- **SQLERRM:** Returns the message associated with the error number

Functions for Trapping Exceptions

Example:

```
DECLARE
    v_error_code          NUMBER;
    v_error_message       VARCHAR2 (255);
BEGIN
    ...
EXCEPTION
    ...
    WHEN OTHERS THEN
        ROLLBACK;
        v_error_code := SQLCODE ;
        v_error_message := SQLERRM ;
        INSERT INTO errors
        VALUES(v_error_code, v_error_message);
END;
```





Trapping User-Defined Exceptions

Declare



Raise



Reference

Name the exception.

Explicitly raise the exception by using the RAISE statement.

Handle the raised exception.



User-Defined Exceptions

Example:

```
DEFINE p_department_desc = 'Information Technology '
DEFINE P_department_number = 300
```

```
DECLARE
    e_invalid_department EXCEPTION;
BEGIN
    UPDATE      departments
    SET         department_name = '&p_department_desc'
    WHERE       department_id = &p_department_number;
    IF SQL%NOTFOUND THEN
        RAISE e_invalid_department;
    END IF;
    COMMIT;
EXCEPTION
    WHEN e_invalid_department THEN
        DBMS_OUTPUT.PUT_LINE('No such department id.');
END;
```

1

2

3



Calling Environments

iSQL*Plus	Displays error number and message to screen
Procedure Builder	Displays error number and message to screen
Oracle Developer Forms	Accesses error number and message in a trigger by means of the <code>ERROR_CODE</code> and <code>ERROR_TEXT</code> packaged functions
Precompiler application	Accesses exception number through the <code>SQLCA</code> data structure
An enclosing PL/SQL block	Traps exception in exception-handling routine of enclosing block



Propagating Exceptions

Subblocks can handle an exception or pass the exception to the enclosing block.

```
DECLARE
    . . .
    e_no_rows      exception;
    e_integrity     exception;
    PRAGMA EXCEPTION_INIT (e_integrity, -2292);

BEGIN
    FOR c_record IN emp_cursor LOOP
        BEGIN
            SELECT . . .
            UPDATE . . .
            IF SQL%NOTFOUND THEN
                RAISE e_no_rows;
            END IF;
        END;
    END LOOP;

    EXCEPTION
        WHEN e_integrity THEN . . .
        WHEN e_no_rows THEN . . .
    END;
```



The RAISE_APPLICATION_ERROR Procedure

Syntax:

```
raise_application_error (error_number,  
                      message[, {TRUE | FALSE}]);
```

- You can use this procedure to issue user-defined error messages from stored subprograms.
- You can report errors to your application and avoid returning unhandled exceptions.

The RAISE_APPLICATION_ERROR Procedure

- **Used in two different places:**
 - Executable section
 - Exception section
- **Returns error conditions to the user in a manner consistent with other Oracle server errors**



RAISE_APPLICATION_ERROR

Executable section:

```
BEGIN  
    ...  
    DELETE FROM employees  
        WHERE manager_id = v_mgr;  
    IF SQL%NOTFOUND THEN  
        RAISE_APPLICATION_ERROR(-20202,  
            'This is not a valid manager');  
    END IF;  
    ...
```

Exception section:

```
...  
EXCEPTION  
    WHEN NO_DATA_FOUND THEN  
        RAISE_APPLICATION_ERROR (-20201,  
            'Manager is not a valid employee.');
```

```
END;
```



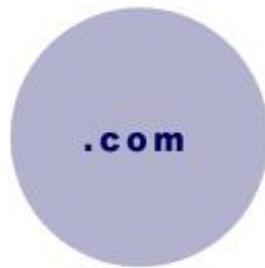
Summary

In this lesson, you should have learned that:

- **Exception types:**
 - Predefined Oracle server error
 - Nonpredefined Oracle server error
 - User-defined error
- **Exception trapping**
- **Exception handling:**
 - Trap the exception within the PL/SQL block.
 - Propagate the exception.



Creating Procedures





Objectives

After completing this lesson, you should be able to do the following:

- **Distinguish anonymous PL/SQL blocks from named PL/SQL blocks (subprograms)**
- **Describe subprograms**
- **List the benefits of using subprograms**
- **List the different environments from which subprograms can be invoked**



Objectives

After completing this lesson, you should be able to do the following:

- **Describe PL/SQL blocks and subprograms**
- **Describe the uses of procedures**
- **Create procedures**
- **Differentiate between formal and actual parameters**
- **List the features of different parameter modes**
- **Create procedures with parameters**
- **Invoke a procedure**
- **Handle exceptions in procedures**
- **Remove a procedure**

PL/SQL Program Constructs

Tools Constructs

- Anonymous blocks
- Application procedures or functions
- Application packages
- Application triggers
- Object types

```
<header> IS | AS
or DECLARE
    • • •
BEGIN
    • • •
EXCEPTION
    • • •
END ;
```

Database Server Constructs

- Anonymous blocks
- Stored procedures or functions
- Stored packages
- Database triggers
- Object types



Overview of Subprograms

A subprogram:

- **Is a named PL/SQL block that can accept parameters and be invoked from a calling environment**
- **Is of two types:**
 - A procedure that performs an action
 - A function that computes a value
- **Is based on standard PL/SQL block structure**
- **Provides modularity, reusability, extensibility, and maintainability**
- **Provides easy maintenance, improved data security and integrity, improved performance, and improved code clarity**



Block Structure for Anonymous PL/SQL Blocks

DECLARE (optional)

Declare PL/SQL objects to be used
within this block

BEGIN (mandatory)

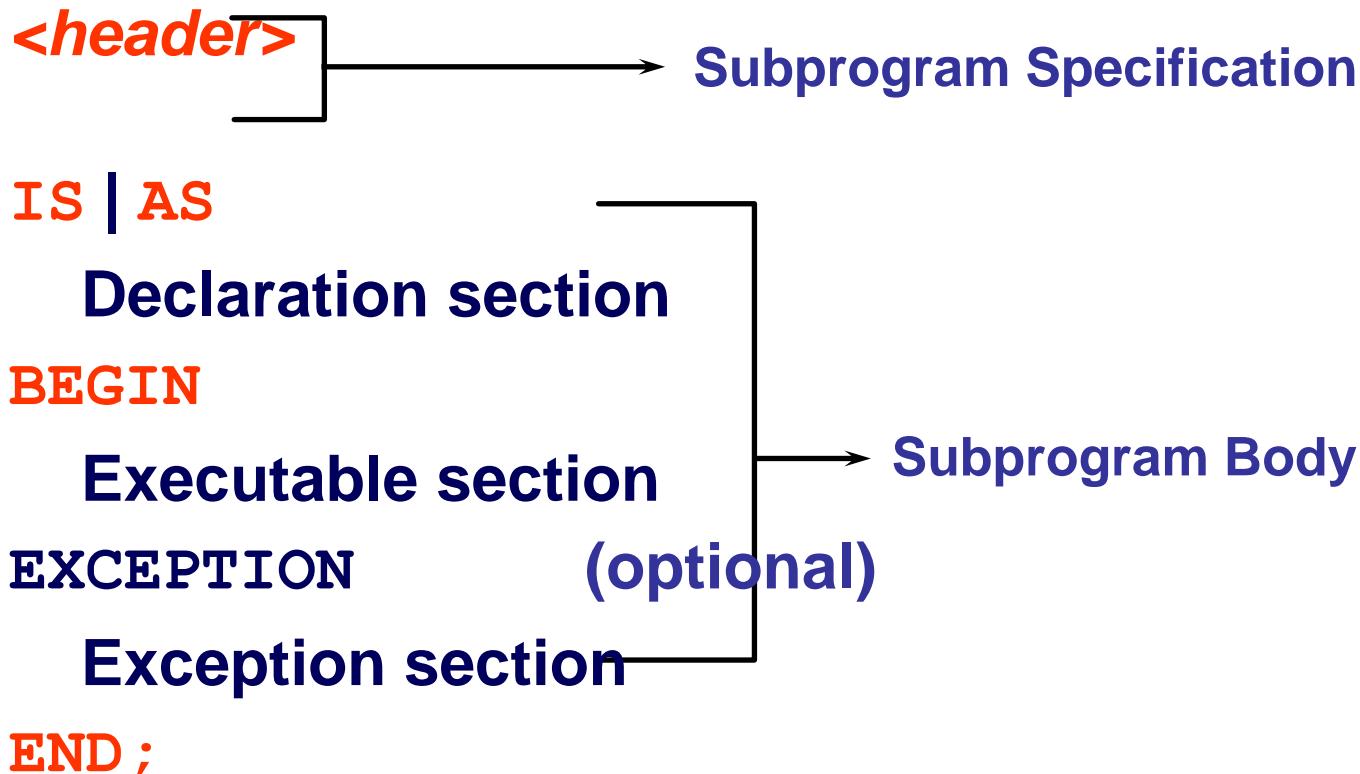
Define the executable statements

EXCEPTION (optional)

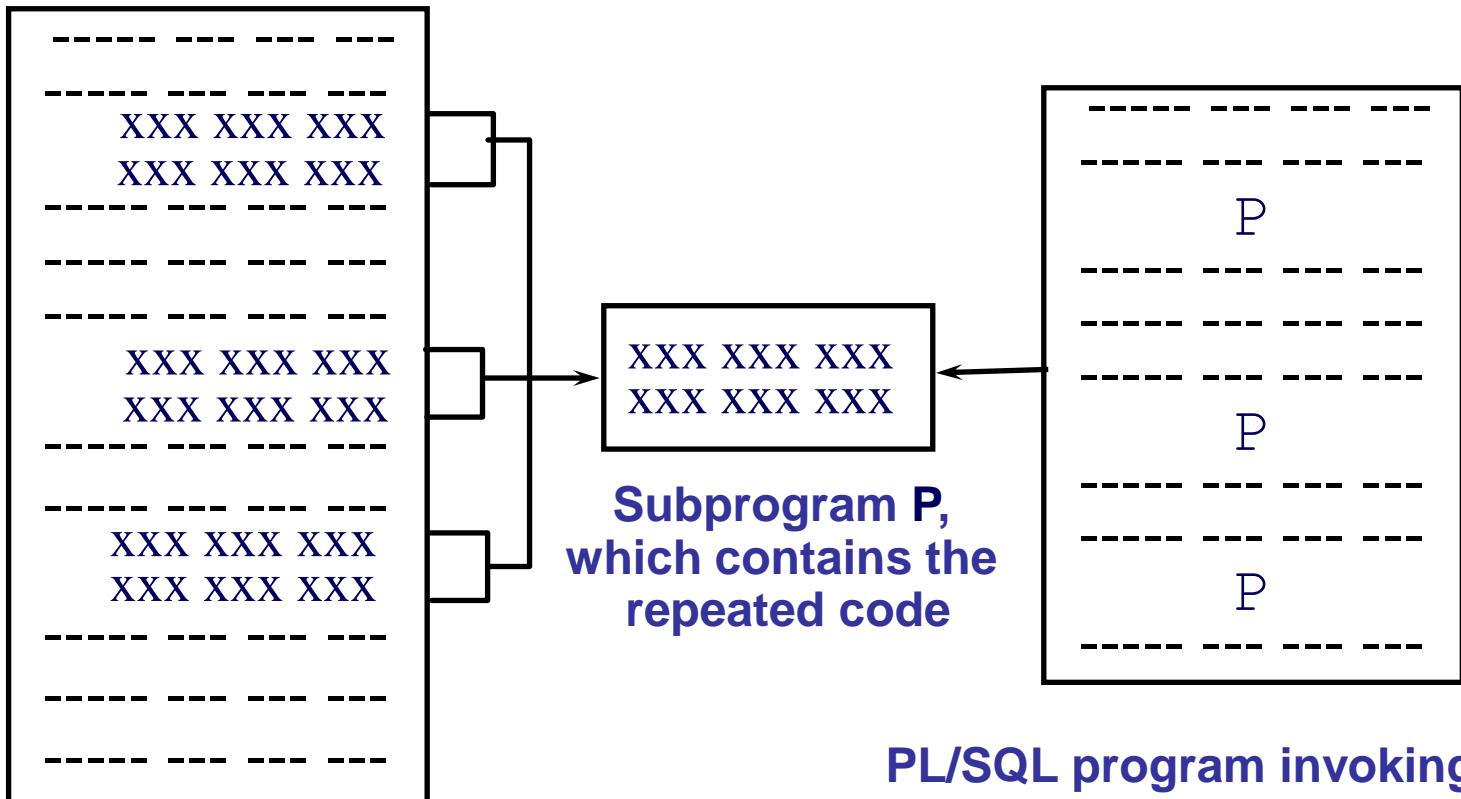
Define the actions that take place if
an error or exception arises

END ; (mandatory)

Block Structure for PL/SQL Subprograms



PL/SQL Subprograms



Code repeated more than once in a PL/SQL program

PL/SQL program invoking the subprogram at multiple locations



Benefits of Subprograms

- Easy maintenance
- Improved data security and integrity
- Improved performance
- Improved code clarity

Invoking Stored Procedures and Functions



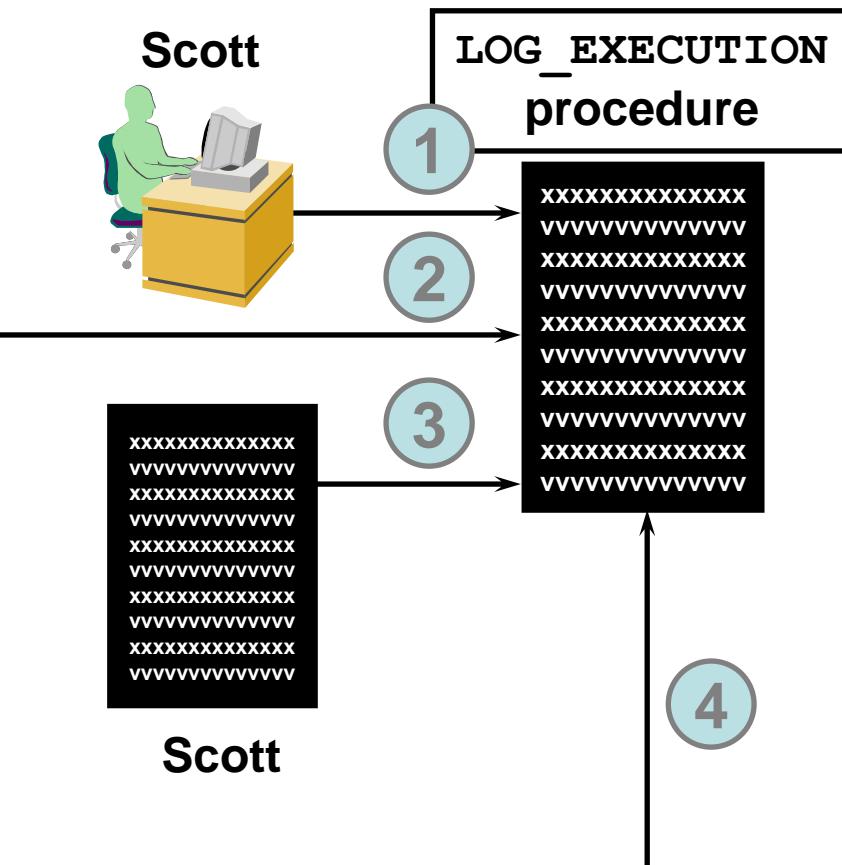
Oracle
Portal



Oracle
Discoverer



Oracle
Forms
Developer





What Is a Procedure?

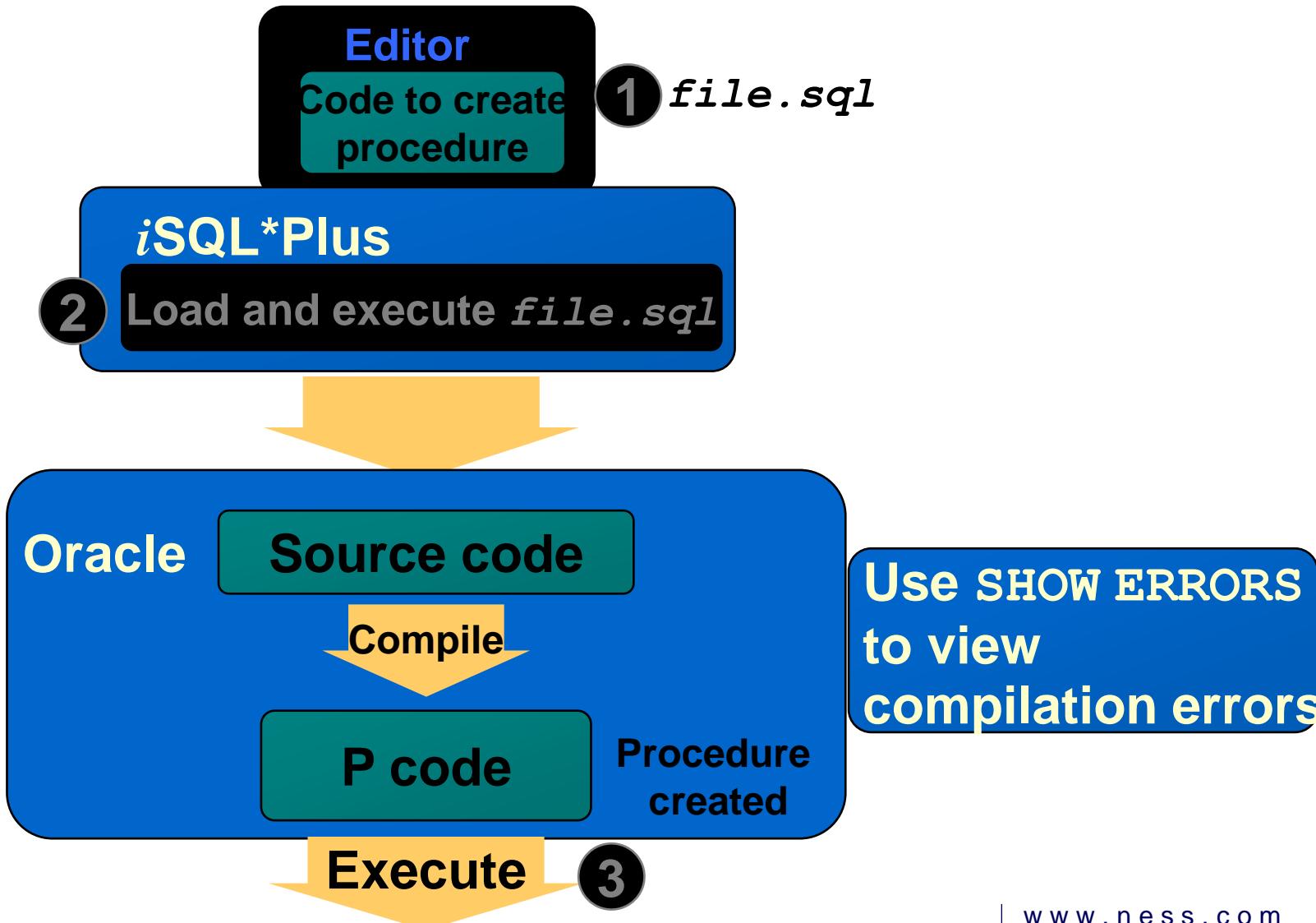
- A procedure is a type of subprogram that performs an action.
- A procedure can be stored in the database, as a schema object, for repeated execution.

Syntax for Creating Procedures

```
CREATE [OR REPLACE] PROCEDURE procedure_name
  [ (parameter1 [mode1] datatype1,
    parameter2 [mode2] datatype2,
    . . . ) ]
IS | AS
PL/SQL Block;
```

- The **REPLACE** option indicates that if the procedure exists, it will be dropped and replaced with the new version created by the statement.
- PL/SQL block starts with either **BEGIN** or the declaration of local variables and ends with either **END** or **END *procedure_name***.

Developing Procedures



Formal Versus Actual Parameters

- **Formal parameters:** variables declared in the parameter list of a subprogram specification

Example:

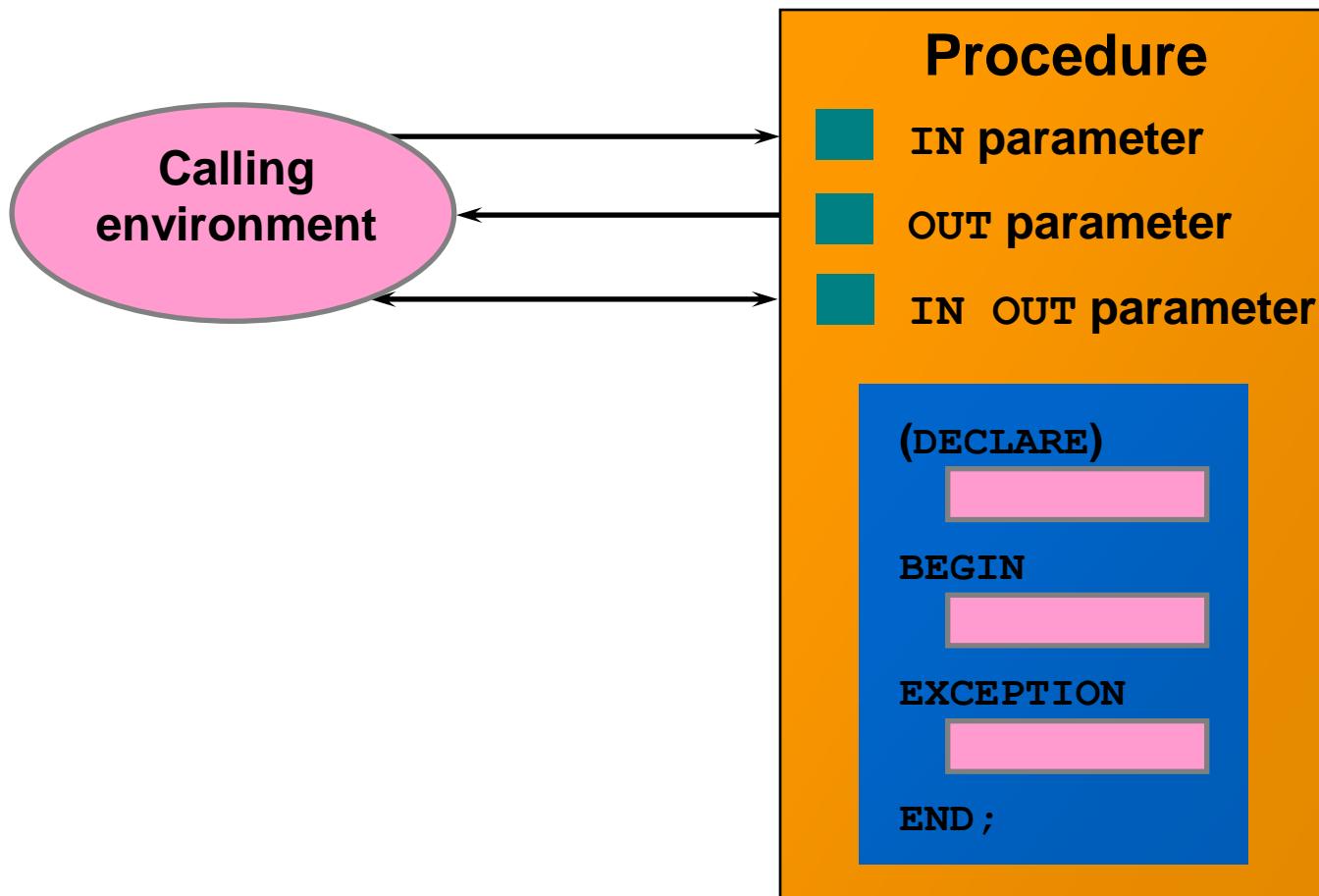
```
CREATE PROCEDURE raise_sal(p_id NUMBER, p_amount NUMBER)
...
END raise_sal;
```

- **Actual parameters:** variables or expressions referenced in the parameter list of a subprogram call

Example:

```
raise_sal(v_id, 2000)
```

Procedural Parameter Modes

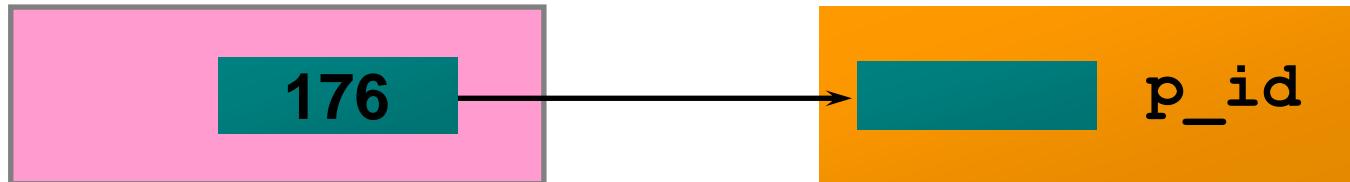


Creating Procedures with Parameters

IN	OUT	IN OUT
Default mode	Must be specified	Must be specified
Value is passed into subprogram	Returned to calling environment	Passed into subprogram; returned to calling environment
Formal parameter acts as a constant	Uninitialized variable	Initialized variable
Actual parameter can be a literal, expression, constant, or initialized variable	Must be a variable	Must be a variable
Can be assigned a default value	Cannot be assigned a default value	Cannot be assigned a default value



IN Parameters: Example

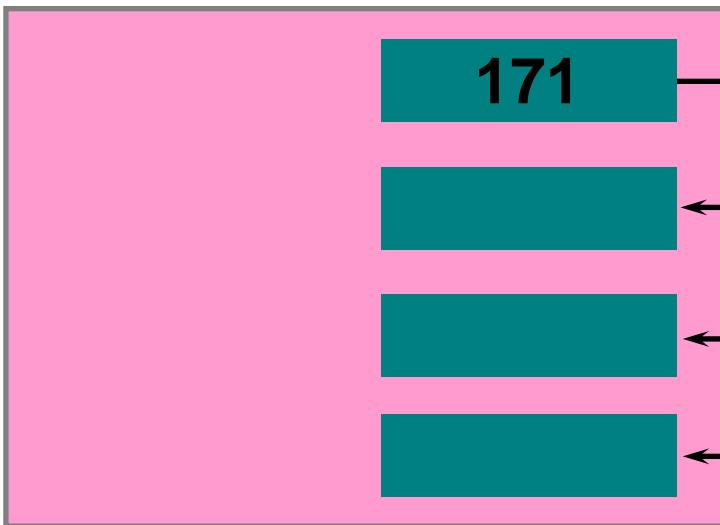


```
CREATE OR REPLACE PROCEDURE raise_salary
    (p_id IN employees.employee_id%TYPE)
IS
BEGIN
    UPDATE employees
    SET salary = salary * 1.10
    WHERE employee_id = p_id;
END raise_salary;
/
```

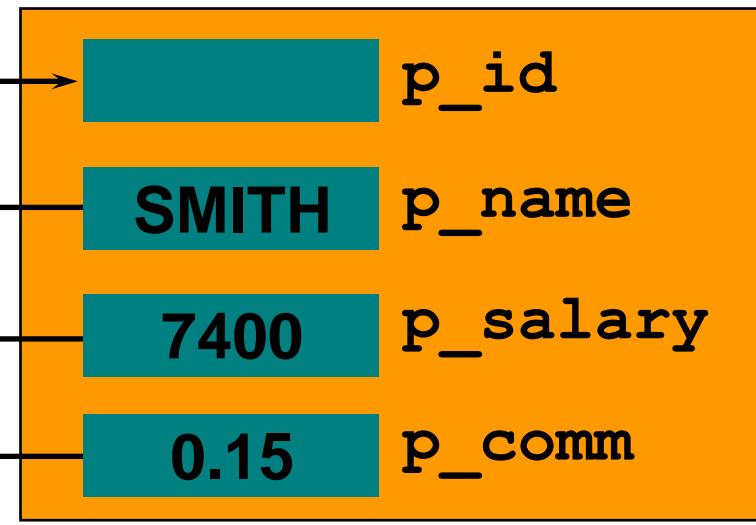
Procedure created.

OUT Parameters: Example

Calling environment



QUERY_EMP procedure



OUT Parameters: Example

emp_query.sql

```
CREATE OR REPLACE PROCEDURE query_emp
    (p_id      IN employees.employee_id%TYPE,
     p_name    OUT employees.last_name%TYPE,
     p_salary  OUT employees.salary%TYPE,
     p_comm    OUT employees.commission_pct%TYPE)
IS
BEGIN
    SELECT last_name, salary, commission_pct
    INTO   p_name, p_salary, p_comm
    FROM   employees
    WHERE  employee_id = p_id;
END query_emp;
/
```

Procedure created.

Viewing OUT Parameters

- Load and run the `emp_query.sql` script file to create the `QUERY_EMP` procedure.
- Declare host variables, execute the `QUERY_EMP` procedure, and print the value of the global `G_NAME` variable.

```
DECLARE
    outputString VARCHAR(25);
BEGIN
    LoginValidation(9531,'priya',
    outputString);
    dbms_output.put_line(outputString);
END;
```

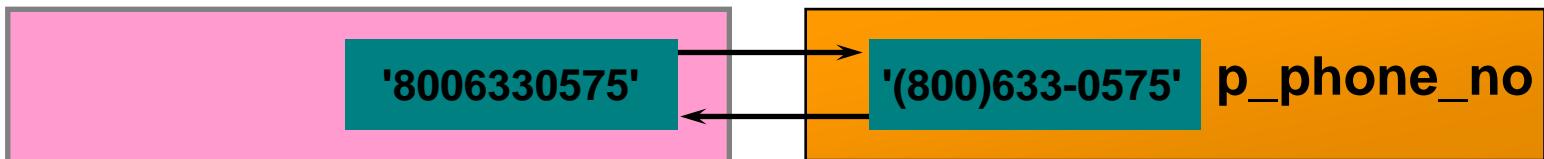
PL/SQL procedure successfully completed.

G_NAME
Smith

IN OUT Parameters

FORMAT_PHONE procedure

Calling environment



```
CREATE OR REPLACE PROCEDURE format_phone
  (p_phone_no IN OUT VARCHAR2)
IS
BEGIN
  p_phone_no := '(' || SUBSTR(p_phone_no,1,3) ||
                 ')' ' || SUBSTR(p_phone_no,4,3) ||
                 '-' || SUBSTR(p_phone_no,7);
END format_phone;
/
```

Procedure created.

Viewing IN OUT Parameters

```
VARIABLE g_phone_no VARCHAR2(15)
BEGIN
    :g_phone_no := '8006330575';
END;
/
PRINT g_phone_no
EXECUTE format_phone (:g_phone_no)
PRINT g_phone_no
```

PL/SQL procedure successfully completed.

G_PHONE_NO
8006330575

PL/SQL procedure successfully completed.

G_PHONE_NO
(800)633-0575



Methods for Passing Parameters

- **Positional:** List actual parameters in the same order as formal parameters.
- **Named:** List actual parameters in arbitrary order by associating each with its corresponding formal parameter.
- **Combination:** List some of the actual parameters as positional and some as named.



DEFAULT Option for Parameters

```
CREATE OR REPLACE PROCEDURE add_dept
  (p_name    IN departments.department_name%TYPE
   DEFAULT 'unknown',
   p_loc      IN departments.location_id%TYPE
   DEFAULT 1700)
IS
BEGIN
  INSERT INTO departments(department_id,
                         department_name, location_id)
  VALUES (departments_seq.NEXTVAL, p_name, p_loc);
END add_dept;
/
```

Procedure created.



Examples of Passing Parameters

```
BEGIN
    add_dept;
    add_dept ('TRAINING', 2500);
    add_dept ( p_loc => 2400, p_name =>'EDUCATION' );
    add_dept ( p_loc => 1200) ;
END;
/
SELECT department_id, department_name, location_id
FROM departments;
```

PL/SQL procedure successfully completed.

DEPARTMENT_ID	DEPARTMENT_NAME	LOCATION_ID
10	Administration	1700
20	Marketing	1800
30	Purchasing	1700
40	Human Resources	2400
...		
290	TRAINING	2500
300	EDUCATION	2400
310	unknown	1200

31 rows selected.



Declaring Subprograms

leave_emp2.sql

```
CREATE OR REPLACE PROCEDURE leave_emp2
    (p_id    IN employees.employee_id%TYPE)
IS
    PROCEDURE log_exec
    IS
    BEGIN
        INSERT INTO log_table (user_id, log_date)
        VALUES (USER, SYSDATE);
    END log_exec;
BEGIN
    DELETE FROM employees
    WHERE employee_id = p_id;
    log_exec;
END leave_emp2;
/
```

Invoking a Procedure from an Anonymous PL/SQL Block

```
DECLARE
    v_id NUMBER := 163;
BEGIN
    raise_salary(v_id);          --invoke procedure
    COMMIT;
    ...
END;
```



Invoking a Procedure from Another Procedure

process_emps.sql

```
CREATE OR REPLACE PROCEDURE process_emps
IS
    CURSOR emp_cursor IS
        SELECT employee_id
        FROM employees;
BEGIN
    FOR emp_rec IN emp_cursor
    LOOP
        raise_salary(emp_rec.employee_id);
    END LOOP;
    COMMIT;
END process_emps;
/
```

Handled Exceptions

Calling procedure

```
PROCEDURE  
PROC1  ...  
IS  
...  
BEGIN  
...  
    PROC2 (arg1);  
...  
EXCEPTION  
...  
END  PROC1;
```

Called procedure

```
PROCEDURE  
PROC2  ...  
IS  
...  
BEGIN  
...  
EXCEPTION  
...  
END  PROC2;
```

Exception raised

Exception handled

Control returns to
calling procedure



Handled Exceptions

```
CREATE PROCEDURE p2_ins_dept(p_locid NUMBER) IS
    v_did NUMBER(4);
BEGIN
    DBMS_OUTPUT.PUT_LINE('Procedure p2_ins_dept started');
    INSERT INTO departments VALUES (5, 'Dept 5', 145, p_locid);
    SELECT department_id INTO v_did FROM employees
        WHERE employee_id = 999;
END;

CREATE PROCEDURE p1_ins_loc(p_lid NUMBER, p_city VARCHAR2)
IS
    v_city VARCHAR2(30); v_dname VARCHAR2(30);
BEGIN
    DBMS_OUTPUT.PUT_LINE('Main Procedure p1_ins_loc');
    INSERT INTO locations (location_id, city) VALUES (p_lid, p_city);
    SELECT city INTO v_city FROM locations WHERE location_id = p_lid;
    DBMS_OUTPUT.PUT_LINE('Inserted city '||v_city);
    DBMS_OUTPUT.PUT_LINE('Invoking the procedure p2_ins_dept ...');

    p2_ins_dept(p_lid);

EXCEPTION
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE('No such dept/loc for any employee');
END;
```

Unhandled Exceptions

Calling procedure

```
PROCEDURE
PROC1  ...
IS
...
BEGIN
...
PROC2 (arg1);
...
EXCEPTION
...
END  PROC1;
```

Called procedure

```
PROCEDURE
PROC2  ...
IS
...
BEGIN
...
EXCEPTION
...
END  PROC2;
```

Exception raised

Exception unhandled

Control returned to
exception section of
calling procedure



Unhandled Exceptions

```
CREATE PROCEDURE p2_noexcep(p_locid NUMBER) IS
    v_did NUMBER(4);
BEGIN
    DBMS_OUTPUT.PUT_LINE('Procedure p2_noexcep started');
    INSERT INTO departments VALUES (6, 'Dept 6', 145, p_locid);
    SELECT department_id INTO v_did FROM employees
        WHERE employee_id = 999;
END;
```

```
CREATE PROCEDURE p1_noexcep(p_lid NUMBER, p_city VARCHAR2)
IS
    v_city VARCHAR2(30); v_dname VARCHAR2(30);
BEGIN
    DBMS_OUTPUT.PUT_LINE(' Main Procedure p1_noexcep');
    INSERT INTO locations (location_id, city) VALUES (p_lid, p_city);
    SELECT city INTO v_city FROM locations WHERE location_id = p_lid;
    DBMS_OUTPUT.PUT_LINE('Inserted new city'||v_city);
    DBMS_OUTPUT.PUT_LINE('Invoking the procedure p2_noexcep ...');
    p2_noexcep(p_lid);
END;
```



Removing Procedures

Drop a procedure stored in the database.

Syntax:

```
DROP PROCEDURE procedure_name
```

Example:

```
DROP PROCEDURE raise_salary;
```

Procedure dropped.



Summary

In this lesson, you should have learned that:

- **A procedure is a subprogram that performs an action.**
- **You create procedures by using the CREATE PROCEDURE command.**
- **You can compile and save a procedure in the database.**
- **Parameters are used to pass data from the calling environment to the procedure.**
- **There are three parameter modes: IN, OUT, and IN OUT .**

Summary

- Local subprograms are programs that are defined within the declaration section of another program.
- Procedures can be invoked from any tool or language that supports PL/SQL.
- You should be aware of the effect of handled and unhandled exceptions on transactions and calling procedures.
- You can remove procedures from the database by using the DROP PROCEDURE command.
- Procedures can serve as building blocks for an application.



Creating Functions

www.



.com





Objectives

After completing this lesson, you should be able to do the following:

- **Describe the uses of functions**
- **Create stored functions**
- **Invoke a function**
- **Remove a function**
- **Differentiate between a procedure and a function**



Overview of Stored Functions

- A function is a named PL/SQL block that returns a value.
- A function can be stored in the database as a schema object for repeated execution.
- A function is called as part of an expression.

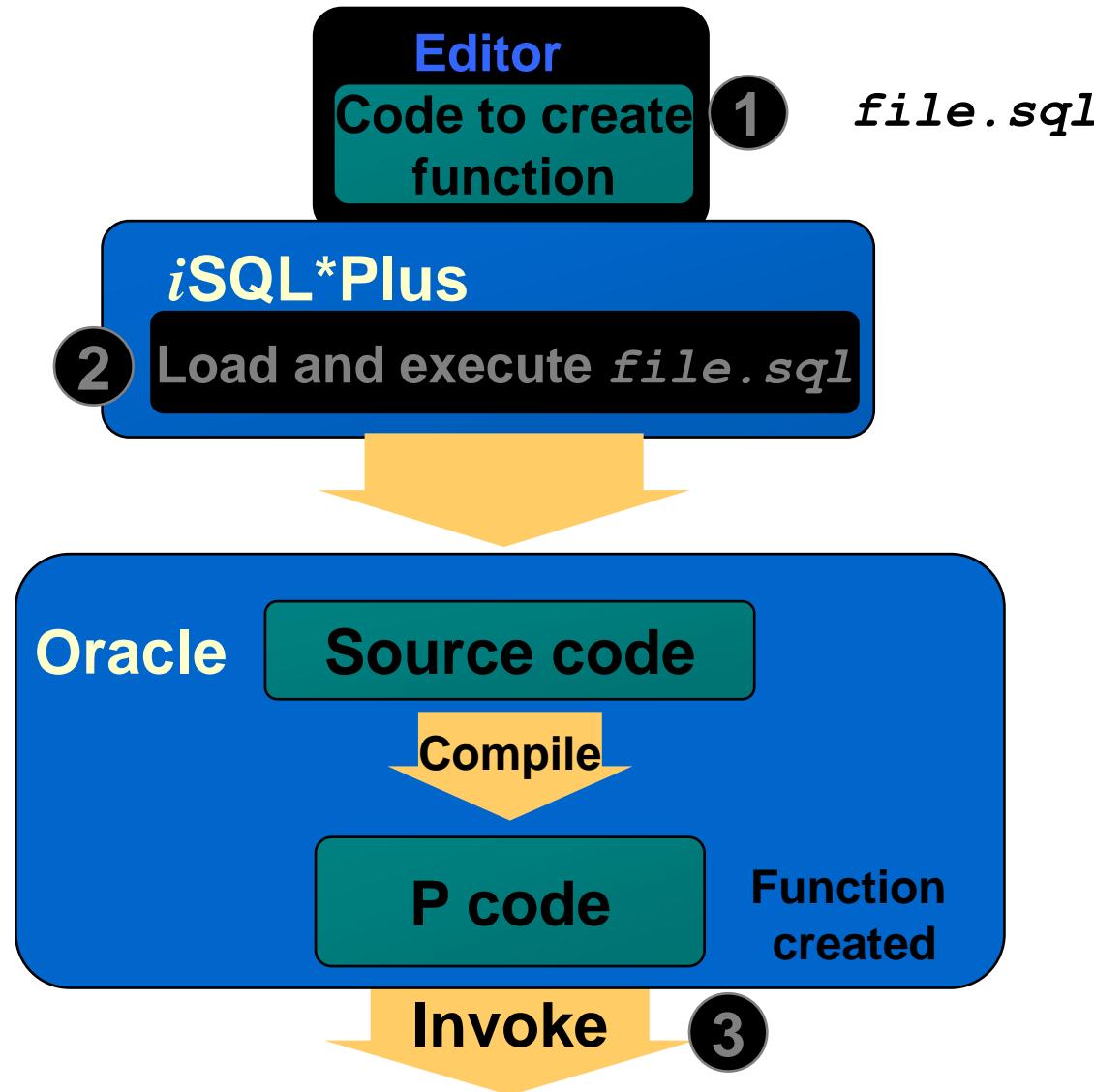


Syntax for Creating Functions

```
CREATE [OR REPLACE] FUNCTION function_name
  [(parameter1 [mode1] datatype1,
    parameter2 [mode2] datatype2,
    . . .)]
RETURN datatype
IS|AS
PL/SQL Block;
```

The PL/SQL block must have at least one RETURN statement.

Creating a Function





Creating a Stored Function by Using *i*SQL*Plus

- 1. Enter the text of the CREATE FUNCTION statement in an editor and save it as a SQL script file.**
- 2. Run the script file to store the source code and compile the function.**
- 3. Use SHOW ERRORS to see compilation errors.**
- 4. When successfully compiled, invoke the function.**

Creating a Stored Function by Using *i*SQL*Plus: Example

get_salary.sql

```
CREATE OR REPLACE FUNCTION get_sal
  (p_id  IN employees.employee_id%TYPE)
  RETURN NUMBER
IS
  v_salary employees.salary%TYPE :=0;
BEGIN
  SELECT salary
  INTO   v_salary
  FROM   employees
  WHERE  employee_id = p_id;
  RETURN v_salary;
END get_sal;
/
```

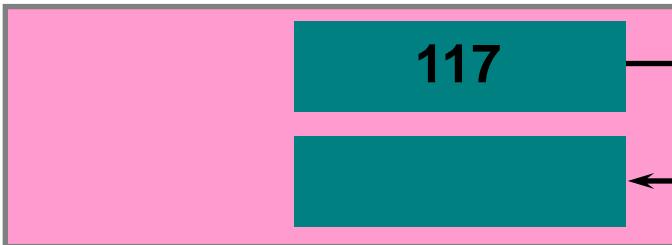


Executing Functions

- **Invoke a function as part of a PL/SQL expression.**
- **Create a variable to hold the returned value.**
- **Execute the function. The variable will be populated by the value returned through a RETURN statement.**

Executing Functions: Example

Calling environment



GET_SAL function



1. Load and run the `get_salary.sql` file to create the function

2 → `VARIABLE g_salary NUMBER`

3 → `EXECUTE :g_salary := get_sal(117)`

4 → `PRINT g_salary`

PL/SQL procedure successfully completed.

G_SALARY
2800

2800



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 Functions in SQL Expressions: Example

```
CREATE OR REPLACE FUNCTION tax(p_value IN NUMBER)
  RETURN NUMBER IS
BEGIN
  RETURN (p_value * 0.08);
END tax;
/
SELECT employee_id, last_name, salary, tax(salary)
FROM   employees
WHERE  department_id = 100;
```

Function created.

EMPLOYEE_ID	LAST_NAME	SALARY	TAX(SALARY)
108	Greenberg	12000	960
109	Faviet	9000	720
110	Chen	8200	656
111	Sciarra	7700	616
112	Urman	7800	624
113	Popp	6900	552

6 rows selected.



Locations to Call User-Defined Functions

- **Select list of a SELECT command**
- **Condition of the WHERE and HAVING clauses**
- **CONNECT BY, START WITH, ORDER BY, and GROUP BY clauses**
- **VALUES clause of the INSERT command**
- **SET clause of the UPDATE command**



Restrictions on Calling Functions from SQL Expressions

To be callable from SQL expressions, a user-defined function must:

- Be a stored function
- Accept only IN parameters
- Accept only valid SQL data types, not PL/SQL specific types, as parameters
- Return data types that are valid SQL data types, not PL/SQL specific types



Restrictions on Calling Functions from SQL Expressions

- Functions called from SQL expressions cannot contain DML statements.
- Functions called from UPDATE/DELETE statements on a table T cannot contain DML on the same table T.
- Functions called from an UPDATE or a DELETE statement on a table T cannot query the same table.
- Functions called from SQL statements cannot contain statements that end the transactions.
- Calls to subprograms that break the previous restriction are not allowed in the function.



Restrictions on Calling from SQL

```
CREATE OR REPLACE FUNCTION dml_call_sql (p_sal NUMBER)
  RETURN NUMBER IS
BEGIN
  INSERT INTO employees(employee_id, last_name, email,
                        hire_date, job_id, salary)
  VALUES (1, 'employee 1', 'empl@company.com',
          SYSDATE, 'SA_MAN', 1000);
  RETURN (p_sal + 100);
END;
/
```

Function created.

```
UPDATE employees SET salary = dml_call_sql(2000)
 WHERE employee_id = 170;
```

```
UPDATE employees SET salary = dml_call_sql(2000)
 *
```

ERROR at line 1:

ORA-04091: table PLSQL.EMPLOYEES is mutating, trigger/function may not see it

ORA-06512: at "PLSQL.DML_CALL_SQL", line 4



Removing Functions

Drop a stored function.

Syntax:

```
DROP FUNCTION function_name
```

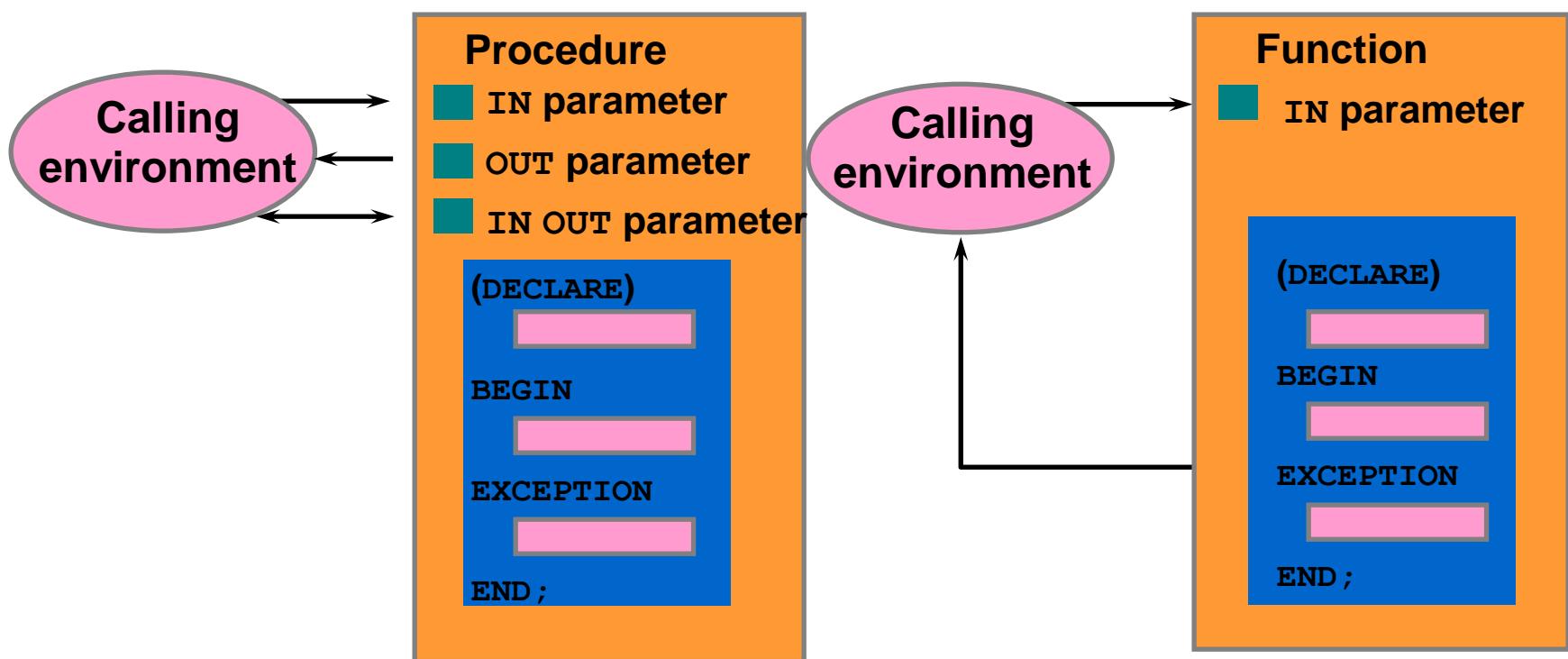
Example:

```
DROP FUNCTION get_sal;
```

Function dropped.

- **All the privileges granted on a function are revoked when the function is dropped.**
- **The CREATE OR REPLACE syntax is equivalent to dropping a function and recreating it. Privileges granted on the function remain the same when this syntax is used.**

Procedure or Function?



Comparing Procedures and Functions

Procedures	Functions
Execute as a PL/SQL statement	Invoke as part of an expression
Do not contain RETURN clause in the header	Must contain a RETURN clause in the header
Can return none, one, or many values	Must return a single value
Can contain a RETURN statement	Must contain at least one RETURN statement



Benefits of Stored Procedures and Functions

- Improved performance
- Easy maintenance
- Improved data security and integrity
- Improved code clarity



Summary

In this lesson, you should have learned that:

- A function is a named PL/SQL block that must return a value.
- A function is created by using the CREATE FUNCTION syntax.
- A function is invoked as part of an expression.
- A function stored in the database can be called in SQL statements.
- A function can be removed from the database by using the DROP FUNCTION syntax.
- Generally, you use a procedure to perform an action and a function to compute a value.



Managing Subprograms





Objectives

After completing this lesson, you should be able to do the following:

- **Contrast system privileges with object privileges**
- **Contrast invokers rights with definers rights**
- **Identify views in the data dictionary to manage stored objects**
- **Describe how to debug subprograms by using the DBMS_OUTPUT package**

Required Privileges

System privileges

DBA grants

```
CREATE (ANY) PROCEDURE
ALTER ANY PROCEDURE
DROP ANY PROCEDURE
EXECUTE ANY PROCEDURE
```

Object privileges

Owner grants

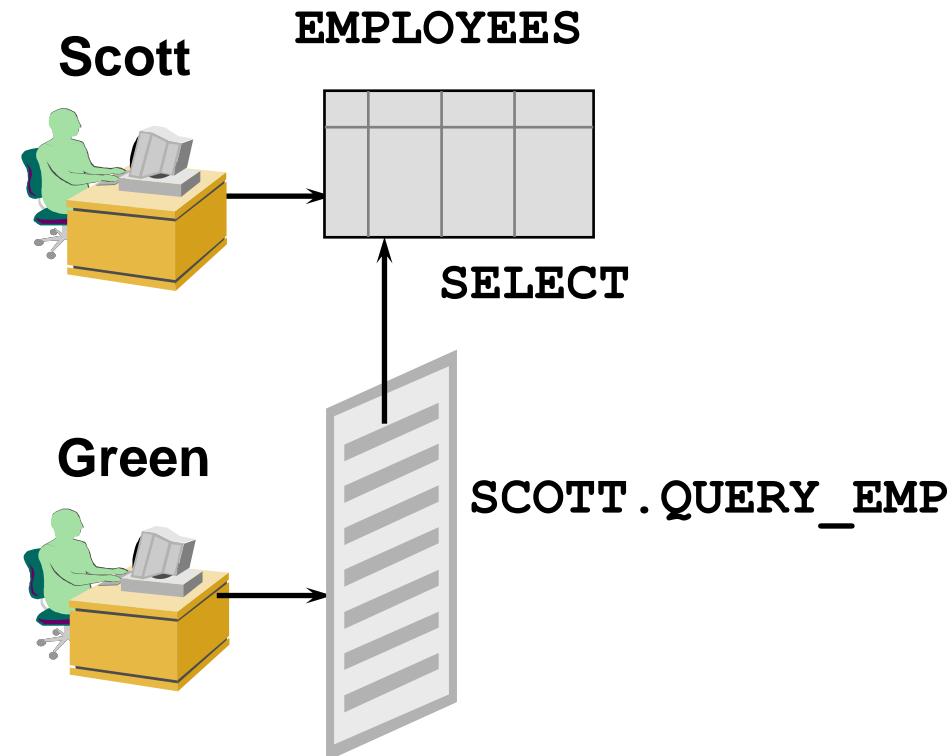
```
EXECUTE
```

To be able to refer and access objects from a different schema in a subprogram, you must be granted access to the referred objects explicitly, not through a role.

Granting Access to Data

Direct access:

```
GRANT SELECT  
ON employees  
TO scott;  
Grant Succeeded.
```



Indirect access:

```
GRANT EXECUTE  
ON query_emp  
TO green;  
Grant Succeeded.
```

The procedure executes with the privileges of the owner (default).

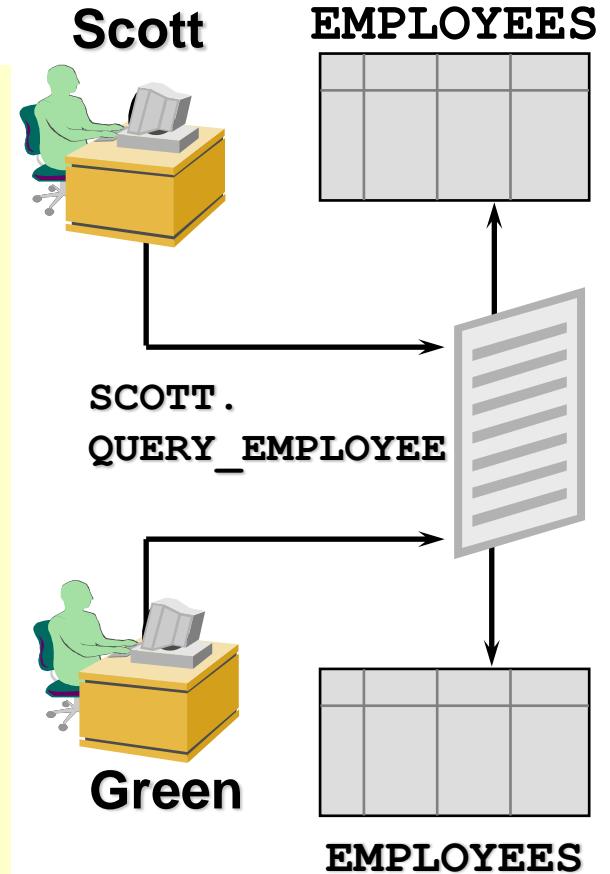
Using Invoker's-Rights

The procedure executes with the privileges of the user.

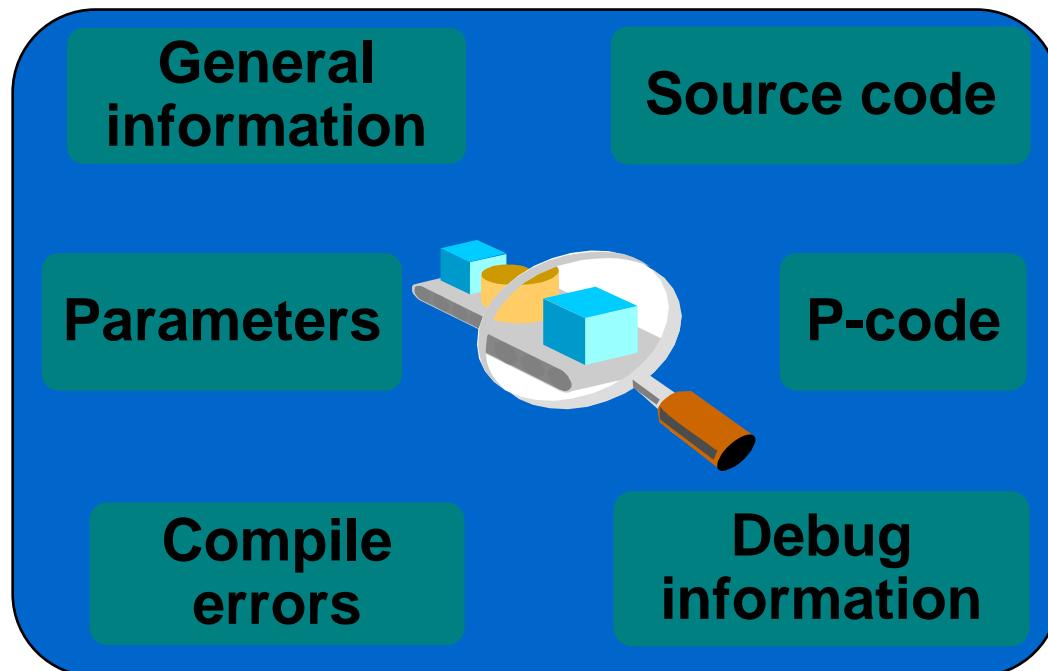
```

CREATE PROCEDURE query_employee
(p_id IN employees.employee_id%TYPE,
 p_name OUT employees.last_name%TYPE,
 p_salary OUT employees.salary%TYPE,
 p_comm OUT
    employees.commission_pct%TYPE)
AUTHID CURRENT_USER
IS
BEGIN
  SELECT last_name, salary,
         commission_pct
    INTO p_name, p_salary, p_comm
   FROM employees
  WHERE employee_id=p_id;
END query_employee;
/

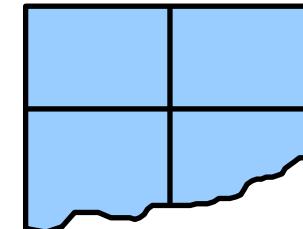
```



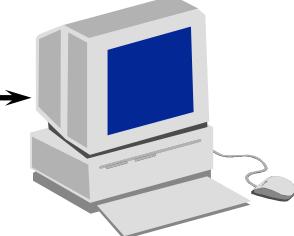
Managing Stored PL/SQL Objects



Data dictionary



Editor



`DESCRIBE ...`

`DBMS_OUTPUT`

USER_OBJECTS

Column	Column Description
OBJECT_NAME	Name of the object
OBJECT_ID	Internal identifier for the object
OBJECT_TYPE	Type of object, for example, TABLE, PROCEDURE, FUNCTION, PACKAGE, PACKAGE BODY, TRIGGER
CREATED	Date when the object was created
LAST_DDL_TIME	Date when the object was last modified
TIMESTAMP	Date and time when the object was last recompiled
STATUS	VALID or INVALID

*Abridged column list



List All Procedures and Functions

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

OBJECT_NAME	OBJECT_TYPE
ADD_DEPT	PROCEDURE
ADD_JOB	PROCEDURE
ADD_JOB_HISTORY	PROCEDURE
ANNUAL_COMP	FUNCTION
DEL_JOB	PROCEDURE
DML CALL SQL	FUNCTION
...	
TAX	FUNCTION
UPD_JOB	PROCEDURE
VALID_DEPTID	FUNCTION

24 rows selected.

USER_SOURCE Data Dictionary View

Column	Column Description
NAME	Name of the object
TYPE	Type of object, for example, PROCEDURE, FUNCTION, PACKAGE, PACKAGE BODY
LINE	Line number of the source code
TEXT	Text of the source code line



List the Code of Procedures and Functions

```
SELECT  text
FROM    user_source
WHERE   name = 'QUERY_EMPLOYEE'
ORDER BY line;
```

TEXT
PROCEDURE query_employee
(p_id IN employees.employee_id%TYPE, p_name OUT employees.last_name%TYPE,
p_salary OUT employees.salary%TYPE, p_comm OUT employees.commission_pct%TYPE)
AUTHID CURRENT_USER
IS
BEGIN
SELECT last_name, salary, commission_pct
INTO p_name,p_salary,p_comm
FROM employees
WHERE employee_id=p_id;
END query_employee;

11 rows selected.

USER_ERRORS

Column	Column Description
NAME	Name of the object
TYPE	Type of object, for example, PROCEDURE, FUNCTION, PACKAGE, PACKAGE BODY, TRIGGER
SEQUENCE	Sequence number, for ordering
LINE	Line number of the source code at which the error occurs
POSITION	Position in the line at which the error occurs
TEXT	Text of the error message

Detecting Compilation Errors: Example

```
CREATE OR REPLACE PROCEDURE log_execution
IS
BEGIN
  INPUT INTO log_table (user_id, log_date)
          -- wrong
  VALUES (USER, SYSDATE);
END;
/
```

Warning: Procedure created with compilation errors.

List Compilation Errors by Using USER_ERRORS

```
SELECT line || '/' || position POS, text
FROM   user_errors
WHERE  name = 'LOG_EXECUTION'
ORDER BY line;
```

POS	TEXT
4/7	PLS-00103: Encountered the symbol "INTO" when expecting one of the following: := . (@ % ;
5/1	PLS-00103: Encountered the symbol "VALUES" when expecting one of the following: . (, % ; limit The symbol "VALUES" was ignored.
6/1	PLS-00103: Encountered the symbol "END"

List Compilation Errors by Using SHOW ERRORS

```
SHOW ERRORS PROCEDURE log_execution
```

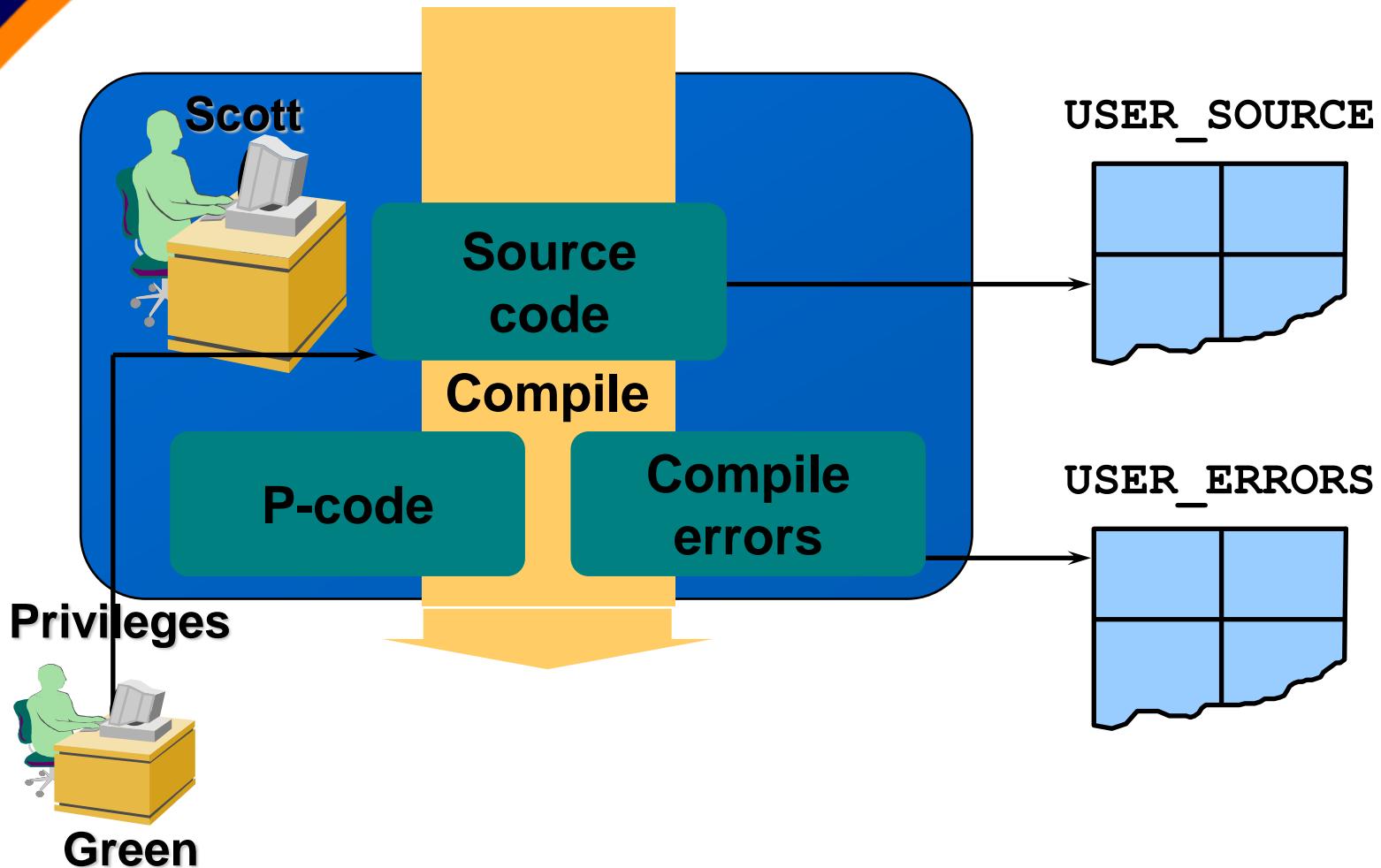
Errors for PROCEDURE LOG_EXECUTION:

LINE/COL	ERROR
4/7	PLS-00103: Encountered the symbol "INTO" when expecting one of the following: := . (@ % ;
5/1	PLS-00103: Encountered the symbol "VALUES" when expecting one of the following: . (, % ; limit The symbol "VALUES" was ignored.
6/1	PLS-00103: Encountered the symbol "END"

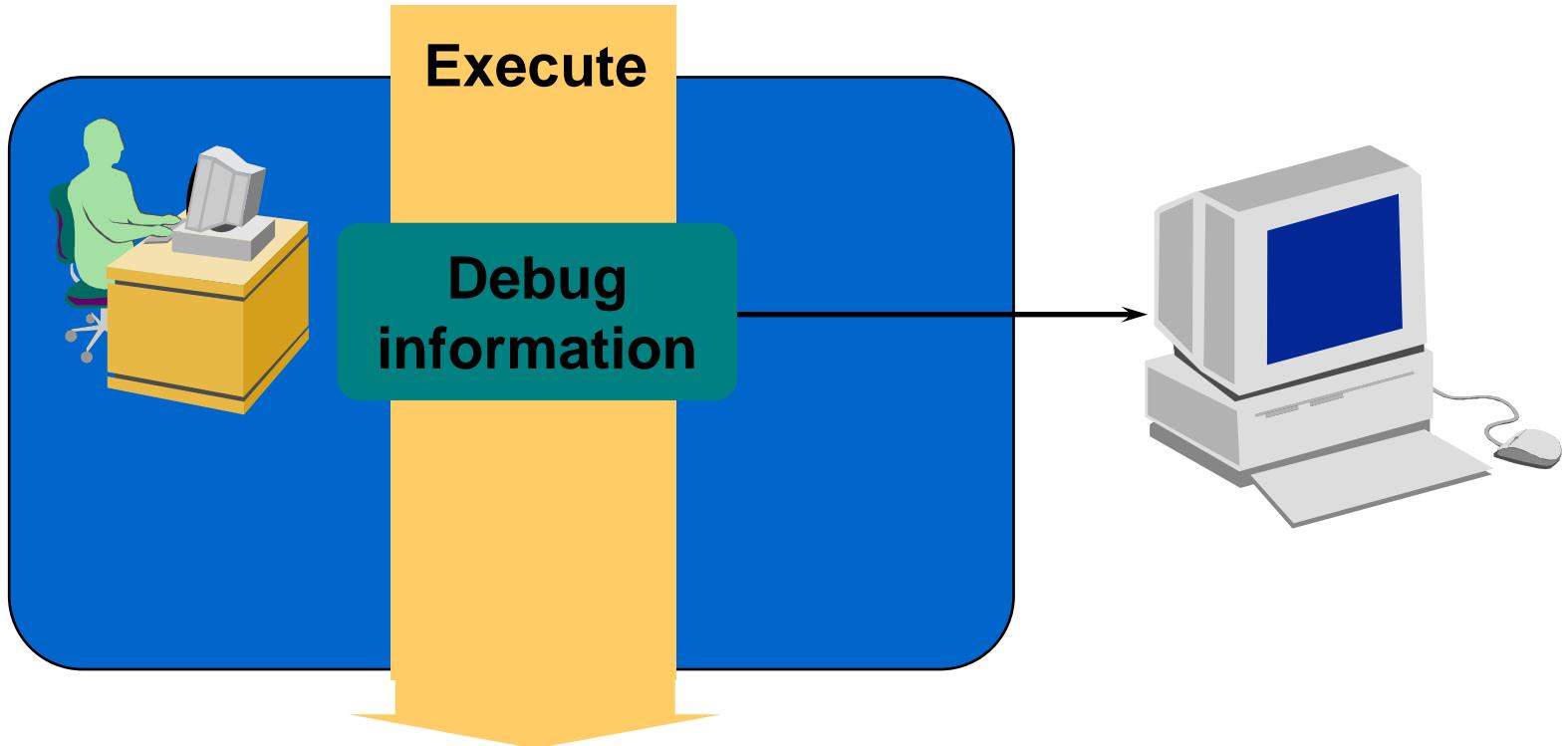
Debugging PL/SQL Program Units

- **The DBMS_OUTPUT package:**
 - Accumulates information into a buffer
 - Allows retrieval of the information from the buffer
- **Autonomous procedure calls (for example, writing the output to a log table)**
- **Software that uses DBMS_DEBUG**
 - Procedure Builder
 - Third-party debugging software

Summary



Summary



We
Are IT



Creating Packages

www.



.com



Objectives

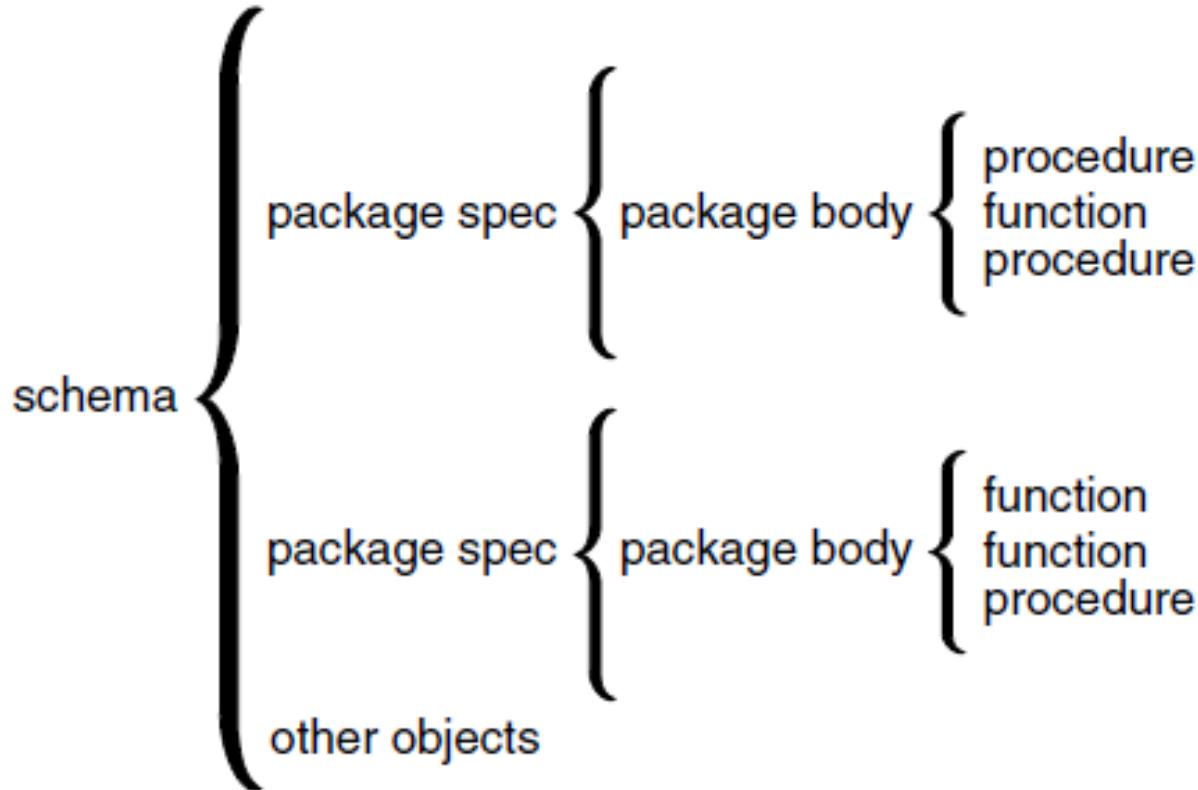
After completing this lesson, you should be able to do the following:

- **Describe packages and list their possible components**
- **Create a package to group together related variables, cursors, constants, exceptions, procedures, and functions**
- **Designate a package construct as either public or private**
- **Invoke a package construct**
- **Describe a use for a bodiless package**

Overview of Packages

Packages:

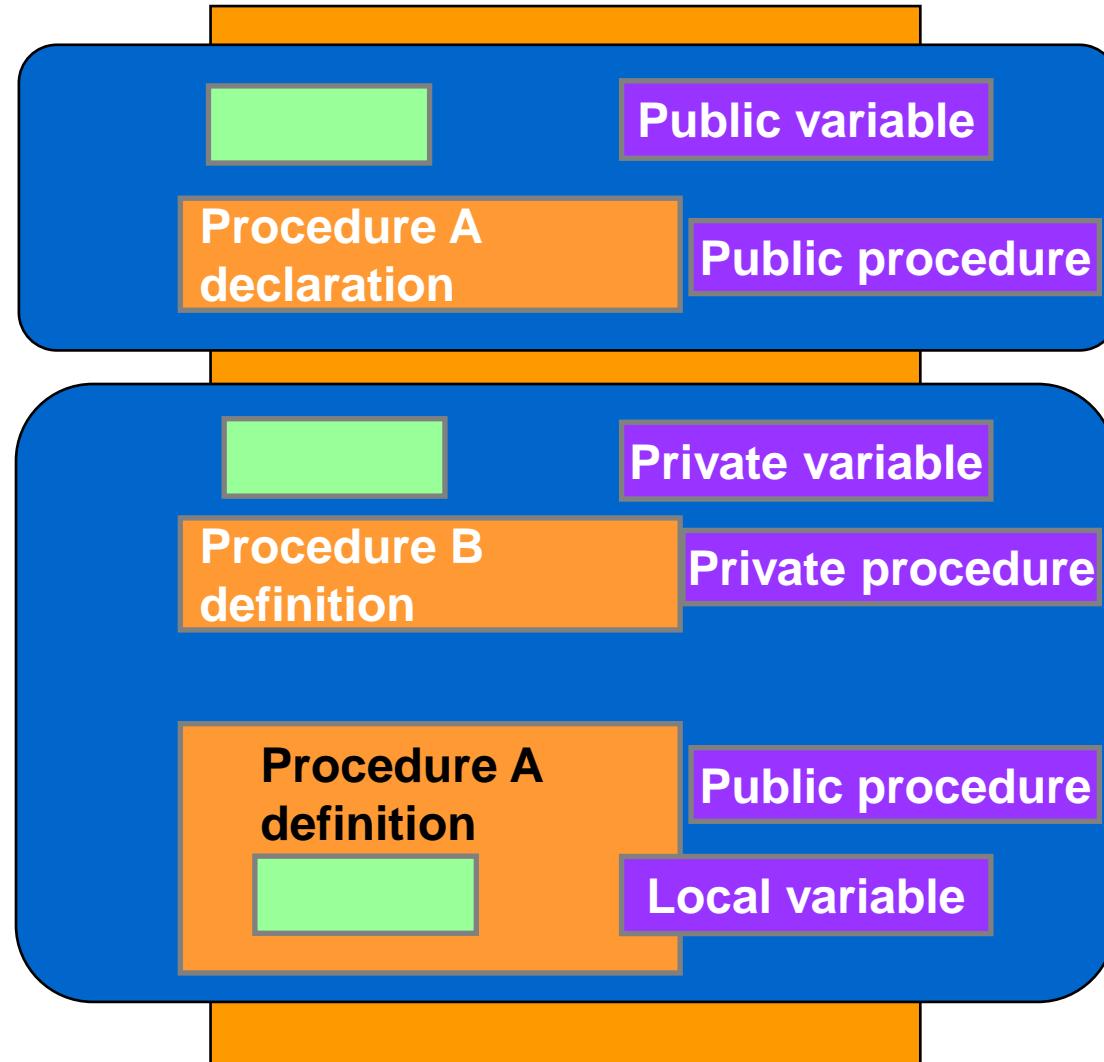
- **Group logically related PL/SQL types, items, and subprograms**
- **Consist of two parts:**
 - Specification
 - Body
- **Cannot be invoked, parameterized, or nested**
- **Allow the Oracle server to read multiple objects into memory at once**



Components of a Package

Package specification

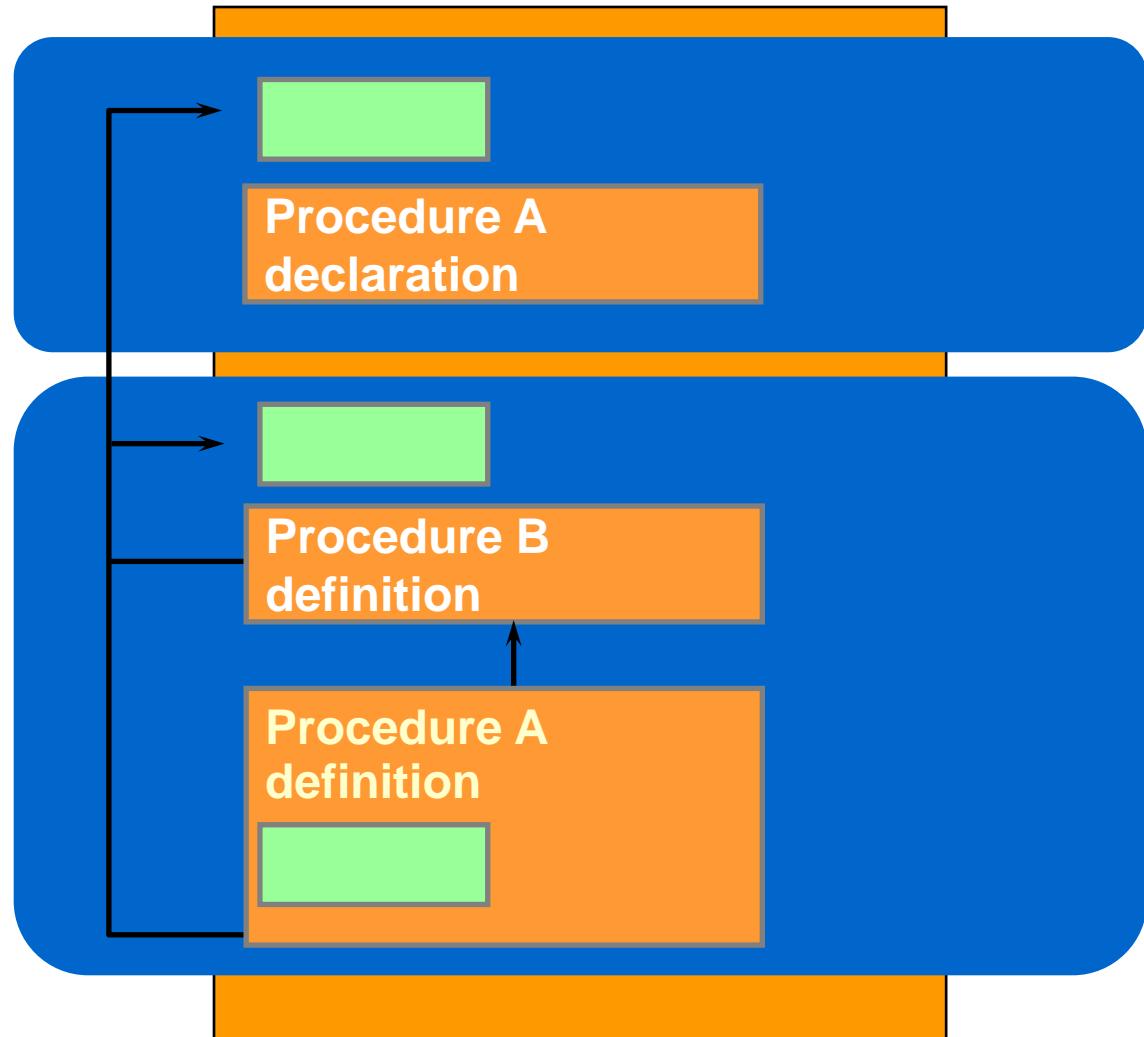
Package body



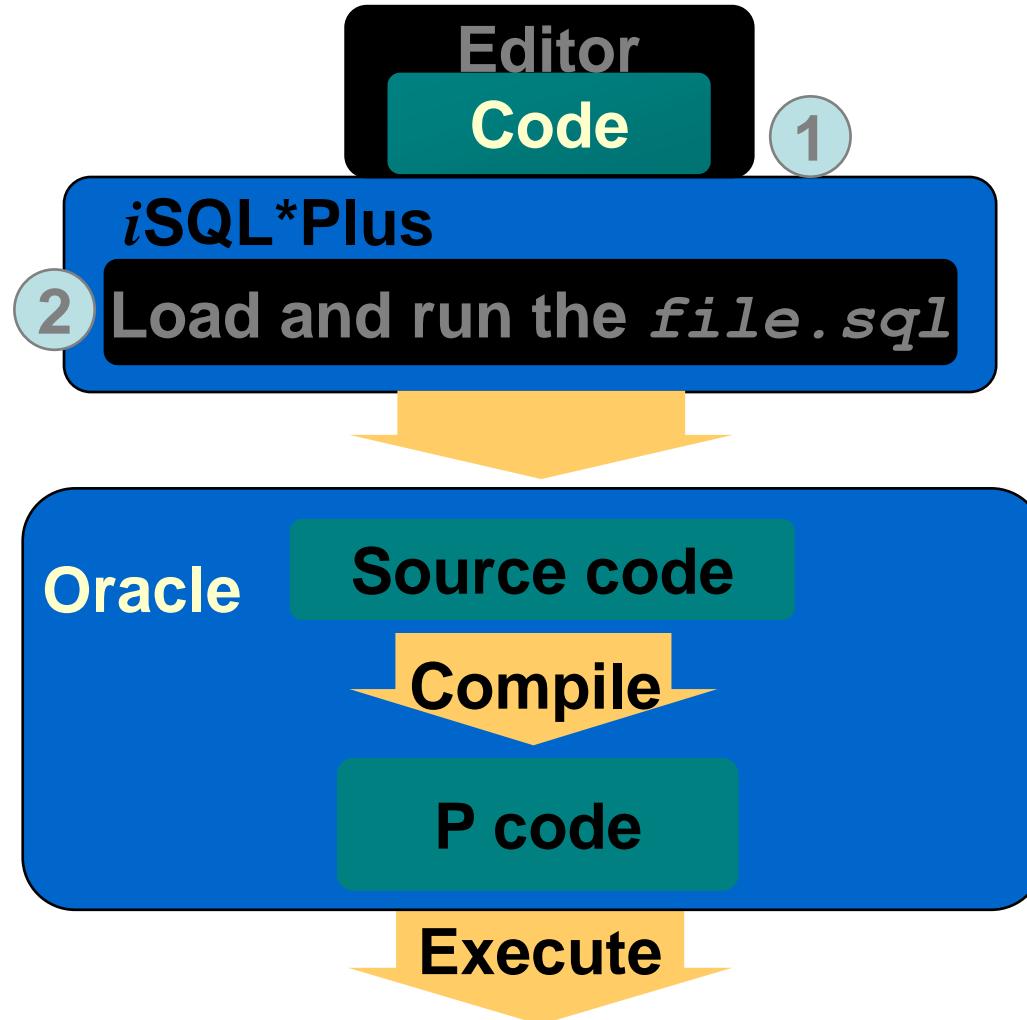
Referencing Package Objects

Package specification

Package body



Developing a Package





Developing a Package

- Saving the text of the CREATE PACKAGE statement in two different SQL files facilitates later modifications to the package.
- 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;
```

- The REPLACE option drops and recreates the package specification.
- Variables declared in the package specification are initialized to NULL by default.
- All the constructs declared in a package specification are visible to users who are granted privileges on the package.

Declaring Public Constructs

Package specification

COMM_PACKAGE package

G_COMM

1

RESET_COMM
procedure
declaration

2

Creating a Package Specification: Example

```
CREATE OR REPLACE PACKAGE comm_package IS
    g_comm NUMBER := 0.10; --initialized to 0.10
    PROCEDURE reset_comm
        (p_comm      IN NUMBER);
END comm_package;
/
```

Package created.

- **G_COMM is a global variable and is initialized to 0.10.**
- **RESET_COMM is a public procedure that is implemented in the package body.**



Creating the Package Body

Syntax:

```
CREATE [OR REPLACE] PACKAGE BODY package_name
IS | AS
    private type and item declarations
    subprogram bodies
END package_name;
```

- The **REPLACE** option drops and recreates the package body.
- Identifiers defined only in the package body are **private constructs**. These are not visible outside the package body.
- All private constructs must be declared before they are used in the public constructs.

Public and Private Constructs

Package specification

Package body

COMM_PACKAGE package

G_COMM

1

RESET_COMM

2

procedure declaration

VALIDATE_COMM

3

function definition

RESET_COMM

2

procedure definition



Creating a Package Body: Example

comm_pack.sql

```
CREATE OR REPLACE PACKAGE BODY comm_package
IS
    FUNCTION validate_comm (p_comm IN NUMBER)
        RETURN BOOLEAN
    IS
        v_max_comm      NUMBER;
    BEGIN
        SELECT      MAX(commission_pct)
        INTO        v_max_comm
        FROM        employees;
        IF      p_comm > v_max_comm THEN RETURN(FALSE) ;
        ELSE      RETURN(TRUE) ;
        END IF;
    END validate_comm;
...
```



Creating a Package Body: Example

comm_pack.sql

```
PROCEDURE  reset_comm (p_comm      IN  NUMBER)
IS
BEGIN
  IF validate_comm(p_comm)
    THEN   g_comm:=p_comm;  --reset global variable
  ELSE
    RAISE_APPLICATION_ERROR(-20210,'Invalid commission');
  END IF;
END reset_comm;
END comm_package;
/
```

Package body created.



Invoking Package Constructs

Example 1: Invoke a function from a procedure within the same package.

```
CREATE OR REPLACE PACKAGE BODY comm_package IS
    . . .
    PROCEDURE reset_comm
        (p_comm    IN NUMBER)
    IS
    BEGIN
        IF validate_comm(p_comm)
        THEN g_comm := p_comm;
        ELSE
            RAISE_APPLICATION_ERROR
                (-20210, 'Invalid commission');
        END IF;
    END reset_comm;
END comm_package;
```



Invoking Package Constructs

Example 2: Invoke a package procedure from iSQL*Plus.

```
EXECUTE comm_package.reset_comm(0.15)
```

Example 3: Invoke a package procedure in a different schema.

```
EXECUTE scott.comm_package.reset_comm(0.15)
```

Example 4: Invoke a package procedure in a remote database.

```
EXECUTE comm_package.reset_comm@ny(0.15)
```



Declaring a Bodiless Package

```
CREATE OR REPLACE PACKAGE global_consts IS
    mile_2_kilo      CONSTANT NUMBER := 1.6093;
    kilo_2_mile      CONSTANT NUMBER := 0.6214;
    yard_2_meter      CONSTANT NUMBER := 0.9144;
    meter_2_yard      CONSTANT NUMBER := 1.0936;
END global_consts;
/
EXECUTE DBMS_OUTPUT.PUT_LINE('20 miles = '||20*
                                global_consts.mile_2_kilo||' km')
```

Package created.

20 miles = 32.186 km

PL/SQL procedure successfully completed.

Referencing a Public Variable from a Stand-Alone Procedure

Example:

```
CREATE OR REPLACE PROCEDURE meter_to_yard
    (p_meter IN NUMBER, p_yard OUT NUMBER)
IS
BEGIN
    p_yard := p_meter * global_consts.meter_2_yard;
END meter_to_yard;
/
VARIABLE yard NUMBER
EXECUTE meter_to_yard (1, :yard)
PRINT yard
```

Procedure created.

PL/SQL procedure successfully completed.

YARD
1.0936



Removing Packages

To remove the package specification and the body, use the following syntax:

```
DROP PACKAGE package_name;
```

To remove the package body, use the following syntax:

```
DROP PACKAGE BODY package_name;
```

Guidelines for Developing Packages

- **Construct packages for general use.**
- **Define the package specification before the body.**
- **The package specification should contain only those constructs that you want to be public.**
- **Place items in the declaration part of the package body when you must maintain them throughout a session or across transactions.**
- **Changes to the package specification require recompilation of each referencing subprogram.**
- **The package specification should contain as few constructs as possible**

Advantages of Packages

- **Modularity:** Encapsulate related constructs.
- **Easier application design:** Code and compile specification and body separately.
- **Hiding information:**
 - Only the declarations in the package specification are visible and accessible to applications.
 - Private constructs in the package body are hidden and inaccessible.
 - All coding is hidden in the package body.

Advantages of Packages

- **Added functionality: Persistency of variables and cursors**
- **Better performance:**
 - The entire package is loaded into memory when the package is first referenced.
 - There is only one copy in memory for all users.
 - The dependency hierarchy is simplified.
- **Overloading: Multiple subprograms of the same name**



Summary

In this lesson, you should have learned how to:

- **Improve organization, management, security, and performance by using packages**
- **Group related procedures and functions together in a package**
- **Change a package body without affecting a package specification**
- **Grant security access to the entire package**



Summary

In this lesson, you should have learned how to:

- **Hide the source code from users**
- **Load the entire package into memory on the first call**
- **Reduce disk access for subsequent calls**
- **Provide identifiers for the user session**

Summary

Command	Task
CREATE [OR REPLACE] PACKAGE	Create (or modify) an existing package specification
CREATE [OR REPLACE] PACKAGE BODY	Create (or modify) an existing package body
DROP PACKAGE	Remove both the package specification and the package body
DROP PACKAGE BODY	Remove the package body only



Overloading

- Enables you to use the same name for different subprograms inside a PL/SQL block, a subprogram, or a package
- Requires the formal parameters of the subprograms to differ in number, order, or data type family
- Enables you to build more flexibility because a user or application is not restricted by the specific data type or number of formal parameters

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



Overloading: Example

over_pack.sql

```
CREATE OR REPLACE PACKAGE over_pack
IS
    PROCEDURE add_dept
        (p_deptno IN departments.department_id%TYPE,
         p_name IN departments.department_name%TYPE
                    DEFAULT 'unknown',
         p_loc IN departments.location_id%TYPE DEFAULT 0);
    PROCEDURE add_dept
        (p_name IN departments.department_name%TYPE
                    DEFAULT 'unknown',
         p_loc IN departments.location_id%TYPE DEFAULT 0);
END over_pack;
/
```

Package created.



Overloading: Example

over_pack_body.sql

```
CREATE OR REPLACE PACKAGE BODY over_pack IS
  PROCEDURE add_dept
    (p_deptno IN departments.department_id%TYPE,
     p_name IN departments.department_name%TYPE DEFAULT 'unknown',
     p_loc  IN departments.location_id%TYPE DEFAULT 0)
  IS
  BEGIN
    INSERT INTO departments (department_id,
                           department_name, location_id)
      VALUES (p_deptno, p_name, p_loc);
  END add_dept;
  PROCEDURE add_dept
    (p_name IN departments.department_name%TYPE DEFAULT 'unknown',
     p_loc  IN departments.location_id%TYPE DEFAULT 0)
  IS
  BEGIN
    INSERT INTO departments (department_id,
                           department_name, location_id)
      VALUES (departments_seq.NEXTVAL, p_name, p_loc);
  END add_dept;
END over_pack;
/
```



Overloading: Example

- Most built-in functions are overloaded.
- For example, see the `TO_CHAR` function of the `STANDARD` package.

```
FUNCTION TO_CHAR (p1 DATE) RETURN VARCHAR2;  
FUNCTION TO_CHAR (p2 NUMBER) RETURN VARCHAR2;  
FUNCTION TO_CHAR (p1 DATE, P2 VARCHAR2) RETURN VARCHAR2;  
FUNCTION TO_CHAR (p1 NUMBER, P2 VARCHAR2) RETURN VARCHAR2;
```

- If you redeclare a built-in subprogram in a PL/SQL program, your local declaration overrides the global declaration.



Using Forward Declarations

You must declare identifiers before referencing them.

```
CREATE OR REPLACE PACKAGE BODY forward_pack
IS
    PROCEDURE award_bonus(. . .)
    IS
    BEGIN
        calc_rating(. . .);          --illegal reference
    END;

    PROCEDURE calc_rating(. . .)
    IS
    BEGIN
        ...
    END;

END forward_pack;
/
```



Using Forward Declarations

```
CREATE OR REPLACE PACKAGE BODY forward_pack
IS

PROCEDURE calc_rating(. . .);      -- forward declaration

PROCEDURE award_bonus(. . .)
IS
BEGIN
calc_rating(. . .);
. . .
END;

PROCEDURE calc_rating(. . .)
IS
BEGIN
. . .
END;

END forward_pack;
/
```



Creating Database Triggers

www.



.com





Objectives

After completing this lesson, you should be able to

do the following:

- **Describe different types of triggers**
- **Describe database triggers and their use**
- **Create database triggers**
- **Describe database trigger firing rules**
- **Remove database triggers**



Types of Triggers

A trigger:

- Is a PL/SQL block or a PL/SQL procedure associated with a table, view, schema, or the database
- Executes implicitly whenever a particular event takes place
- Can be either:
 - Application trigger: Fires whenever an event occurs with a particular application
 - Database trigger: Fires whenever a data event (such as DML) or system event (such as logon or shutdown) occurs on a schema or database

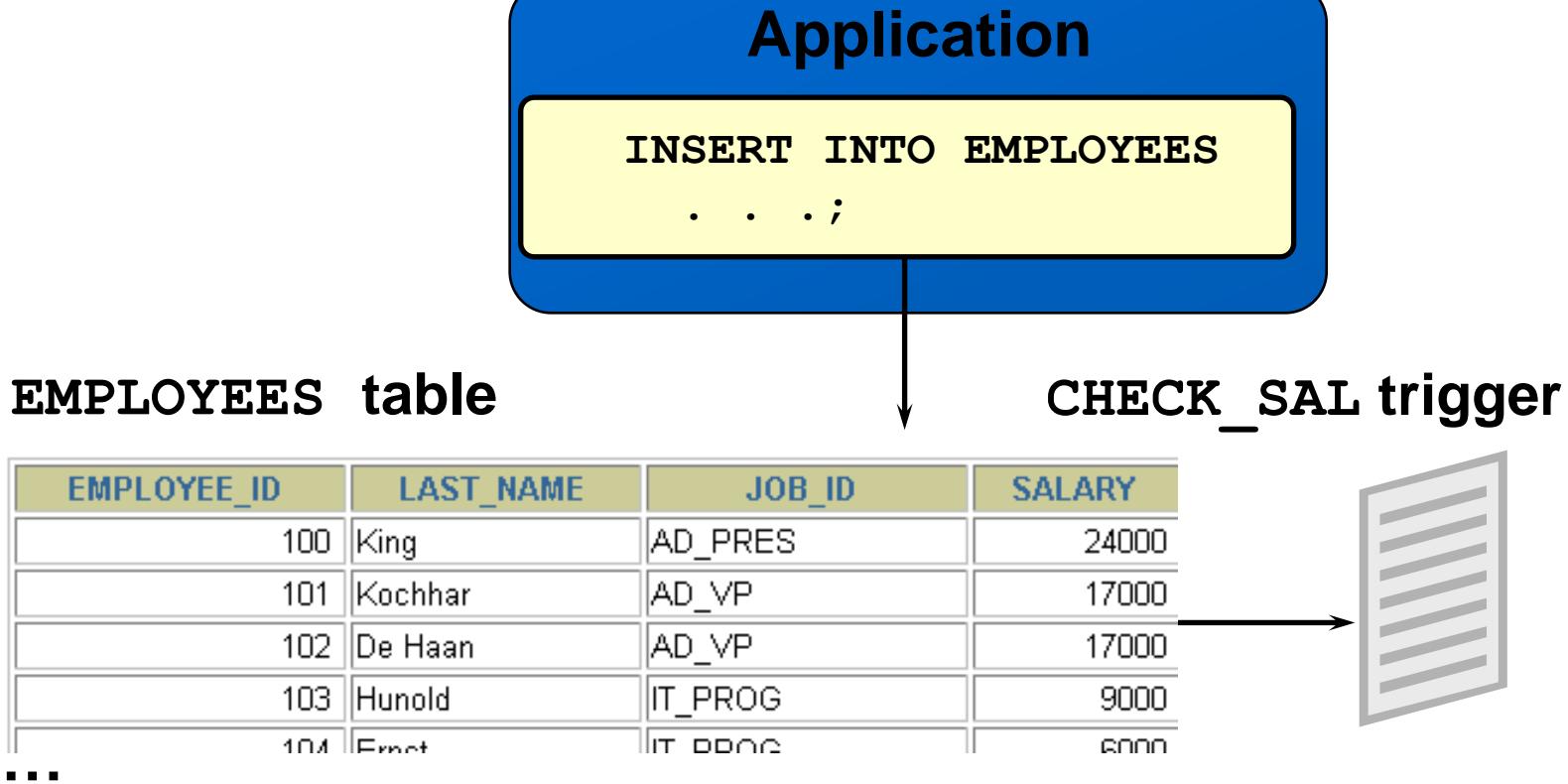


Guidelines for Designing Triggers

- **Design triggers to:**
 - Perform related actions
 - Centralize global operations
- **Do not design triggers:**
 - Where functionality is already built into the Oracle server
 - That duplicate other triggers
- **Create stored procedures and invoke them in a trigger, if the PL/SQL code is very lengthy.**
- **The excessive use of triggers can result in complex interdependencies, which may be difficult to maintain in large applications.**



Database Trigger: Example





Creating DML Triggers

A triggering statement contains:

- **Trigger timing**
 - For table: BEFORE, AFTER
 - For view: INSTEAD OF
- **Triggering event: INSERT, UPDATE, or DELETE**
- **Table name: On table, view**
- **Trigger type: Row or statement**
- **WHEN clause: Restricting condition**
- **Trigger body: PL/SQL block**



DML Trigger Components

Trigger timing: When should the trigger fire?

- **BEFORE:** Execute the trigger body before the triggering DML event on a table.
- **AFTER:** Execute the trigger body after the triggering DML event on a table.
- **INSTEAD OF:** Execute the trigger body instead of the triggering statement. This is used for views that are not otherwise modifiable.



DML Trigger Components

Triggering user event: Which DML statement causes the trigger to execute? You can use any of the following:

- **INSERT**
- **UPDATE**
- **DELETE**



DML Trigger Components

Trigger type: Should the trigger body execute for each row the statement affects or only once?

- **Statement:** The trigger body executes once for the triggering event. This is the default. A statement trigger fires once, even if no rows are affected at all.
- **Row:** The trigger body executes once for each row affected by the triggering event. A row trigger is not executed if the triggering event affects no rows.



DML Trigger Components

Trigger body: What action should the trigger perform?

The trigger body is a PL/SQL block or a call to a procedure.

Firing Sequence

Use the following firing sequence for a trigger on a table, when a single row is manipulated:

DML statement

```
INSERT INTO departments (department_id,  
                        department_name, location_id)  
VALUES (400, 'CONSULTING', 2400);
```

1 row created.

Triggering action

DEPARTMENT_ID	DEPARTMENT_NAME	LOCATION_ID
10	Administration	1700
20	Marketing	1800
30	Purchasing	1700
...		
400	CONSULTING	2400

→ BEFORE statement trigger

...		
400	CONSULTING	2400

→ BEFORE row trigger

→ AFTER row trigger

→ AFTER statement trigger

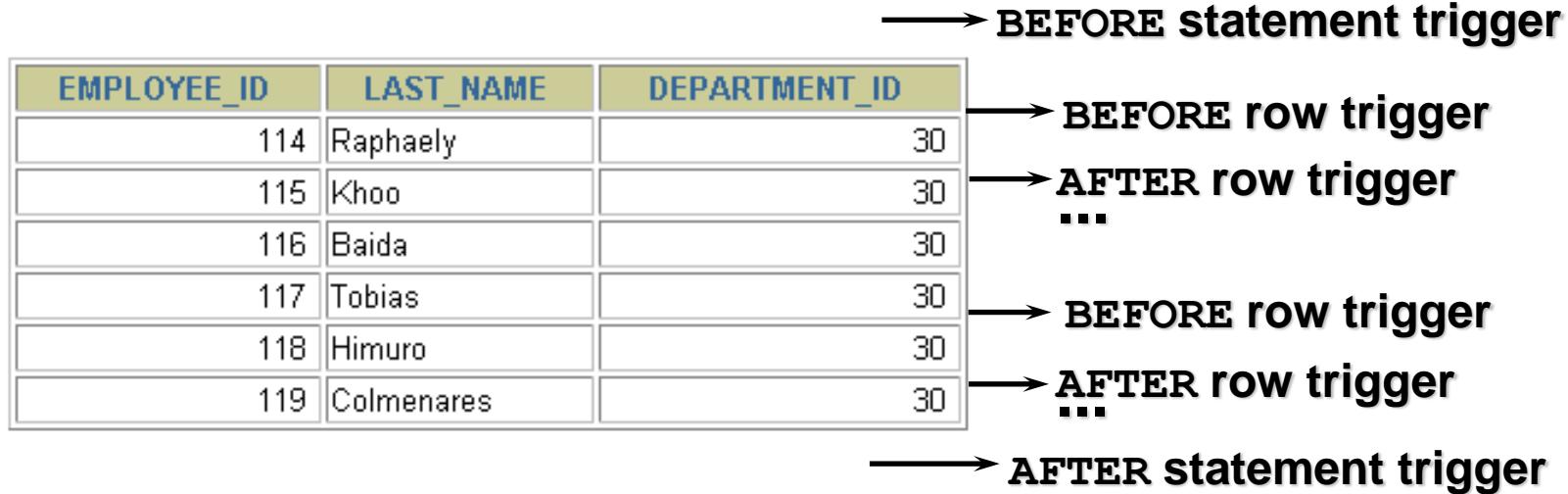


Firing Sequence

Use the following firing sequence for a trigger on a table, when many rows are manipulated:

```
UPDATE employees  
    SET salary = salary * 1.1  
 WHERE department_id = 30;
```

6 rows updated.





Syntax for Creating DML Statement Triggers

Syntax:

```
CREATE [OR REPLACE] TRIGGER trigger_name
    timing
        event1 [OR event2 OR event3]
        ON table_name
trigger_body
```

Note: Trigger names must be unique with respect to other triggers in the same schema.



Creating DML Statement Triggers

Example:

```
CREATE OR REPLACE TRIGGER secure_emp
BEFORE INSERT ON employees
BEGIN
    IF (TO_CHAR(SYSDATE, 'DY') IN ('SAT', 'SUN')) OR
        (TO_CHAR(SYSDATE, 'HH24:MI')
            NOT BETWEEN '08:00' AND '18:00')
    THEN RAISE_APPLICATION_ERROR (-20500, 'You may
                                    insert into EMPLOYEES table only
                                    during business hours.');
    END IF;
END;
/
```

Trigger created.



Testing SECURE_EMP

```
INSERT INTO employees (employee_id, last_name,
                      first_name, email, hire_date,
                      job_id, salary, department_id)
VALUES (300, 'Smith', 'Rob', 'RSMITH', SYSDATE,
        'IT_PROG', 4500, 60);
```

```
INSERT INTO employees (employee_id, last_name, first_name, email,
                      *)
```

ERROR at line 1:

ORA-20500: You may insert into EMPLOYEES table only during business hours.

ORA-06512: at "PLSQL.SECURE_EMP", line 4

ORA-04088: error during execution of trigger 'PLSQL.SECURE_EMP'

```
CREATE OR REPLACE TRIGGER secure_emp
BEFORE INSERT OR UPDATE OR DELETE ON employees
BEGIN
  IF (TO_CHAR (SYSDATE, 'DY') IN ('SAT', 'SUN')) OR
    (TO_CHAR (SYSDATE, 'HH24') NOT BETWEEN '08' AND '18')
  THEN
    IF DELETING THEN
      RAISE_APPLICATION_ERROR (-20502,'You may delete from
                                EMPLOYEES table only during business hours.');
    ELSIF INSERTING THEN
      RAISE_APPLICATION_ERROR (-20500,'You may insert into
                                EMPLOYEES table only during business hours.');
    ELSIF UPDATING ('SALARY') THEN
      RAISE_APPLICATION_ERROR (-20503,'You may update
                                SALARY only during business hours.');
    ELSE
      RAISE_APPLICATION_ERROR (-20504,'You may update
                                EMPLOYEES table only during normal hours.');
    END IF;
  END IF;
END;
```



Creating a DML Row Trigger

Syntax:

```
CREATE [OR REPLACE] TRIGGER trigger_name
    timing
        event1 [OR event2 OR event3]
        ON table_name
        [REFERENCING OLD AS old | NEW AS new]
FOR EACH ROW
    [WHEN (condition) ]
trigger_body
```



Creating DML Row Triggers

```
CREATE OR REPLACE TRIGGER restrict_salary
    BEFORE INSERT OR UPDATE OF salary ON employees
    FOR EACH ROW
    BEGIN
        IF NOT (:NEW.job_id IN ('AD_PRES', 'AD_VP'))
            AND :NEW.salary > 15000
        THEN
            RAISE_APPLICATION_ERROR (-20202, 'Employee
                                    cannot earn this amount');
        END IF;
    END;
/
```

Trigger created.



Using OLD and NEW Qualifiers

```
CREATE OR REPLACE TRIGGER audit_emp_values
AFTER DELETE OR INSERT OR UPDATE ON employees
FOR EACH ROW
BEGIN
    INSERT INTO audit_emp_table (user_name, timestamp,
        id, old_last_name, new_last_name, old_title,
        new_title, old_salary, new_salary)
    VALUES (USER, SYSDATE, :OLD.employee_id,
        :OLD.last_name, :NEW.last_name, :OLD.job_id,
        :NEW.job_id, :OLD.salary, :NEW.salary );
END ;
/
```

Trigger created.



Using OLD and NEW Qualifiers: Example Using Audit Emp Table

```
INSERT INTO employees
    (employee_id, last_name, job_id, salary, ...)
VALUES (999, 'Temp emp', 'SA REP', 1000, ...);
```

```
UPDATE employees
SET salary = 2000, last_name = 'Smith'
WHERE employee_id = 999;
```

1 row created.

1 row updated.

```
SELECT user_name, timestamp, ... FROM audit_emp_table
```

USER_NAME	TIMESTAMP	ID	OLD_LAST_N	NEW_LAST_N	OLD_TITLE	NEW_TITLE	OLD_SALARY	NEW_SALARY
PLSQL	28-SEP-01		Temp emp		SA REP			1000
PLSQL	28-SEP-01	999	Temp emp	Smith	SA REP	SA REP	1000	2000

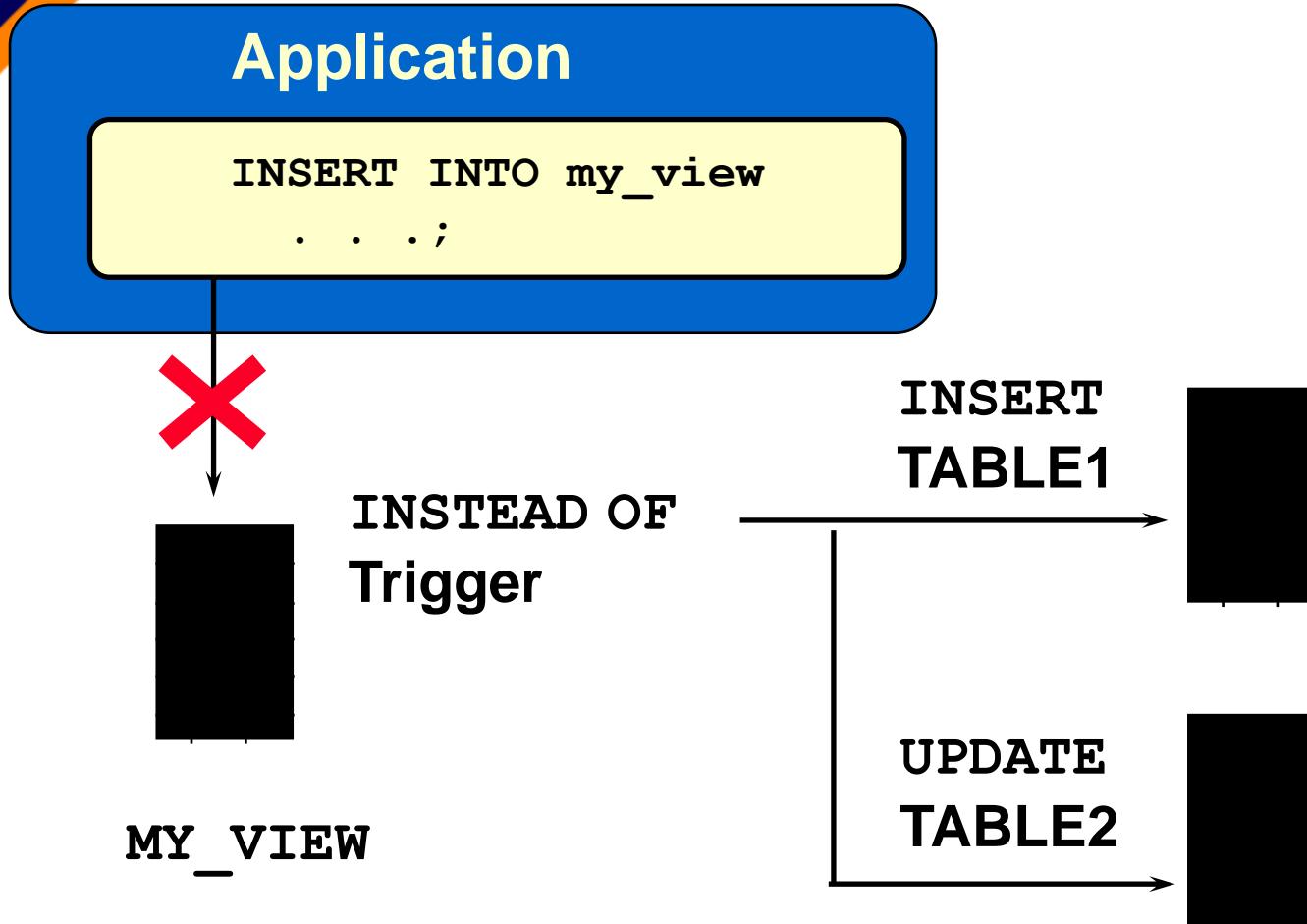


Restricting a Row Trigger

```
CREATE OR REPLACE TRIGGER derive_commission_pct
BEFORE INSERT OR UPDATE OF salary ON employees
FOR EACH ROW
WHEN (NEW.job_id = 'SA_REP')
BEGIN
  IF INSERTING
    THEN :NEW.commission_pct := 0;
  ELSIF :OLD.commission_pct IS NULL
    THEN :NEW.commission_pct := 0;
  ELSE
    :NEW.commission_pct := :OLD.commission_pct + 0.05;
  END IF;
END ;
/
```

Trigger created.

INSTEAD OF Triggers





Creating an INSTEAD OF Trigger

Syntax:

```
CREATE [OR REPLACE] TRIGGER trigger_name
INSTEAD OF
    event1 [OR event2 OR event3]
    ON view_name
    [REFERENCING OLD AS old | NEW AS new]
[FOR EACH ROW]
trigger_body
```



Creating an INSTEAD OF Trigger

INSERT into EMP_DETAILS that is based on EMPLOYEES and DEPARTMENTS tables

1 **INSERT INTO emp_details(employee_id, ...)
VALUES(9001, 'ABBOTT', 3000, 10, 'abbott.mail.com', 'HR_MAN');**

**INSTEAD OF INSERT
into EMP_DETAILS →**

EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID	EMAIL	JOB_ID
100	King	90	SKING	AD_PRES
101	Kochhar	90	NKOCHHAR	AD_VP
102	De Haan	90	LDEHAAN	AD_VP
...				

Creating an INSTEAD OF Trigger

INSERT into EMP_DETAILS that is based on EMPLOYEES and DEPARTMENTS tables

1

```
INSERT INTO emp_details(employee_id, ... )
VALUES(9001, 'ABBOTT', 3000, 10, 'abbott.mail.com', 'HR_MAN');
```

INSTEAD OF INSERT
into EMP_DETAILS —→

EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID	EMAIL	JOB_ID
100	King	90	SKING	AD_PRES
101	Kochhar	90	NKOCHHAR	AD_VP
102	De Haan	90	LDEHAAN	AD_VP
...				

2

INSERT into
NEW_EMPS

EMPLOYEE_ID	LAST_NAME	SALARY	DEPARTMENT_ID	EMAIL
100	King	24000	90	SKING
101	Kochhar	17000	90	NKOCHHAR
102	De Haan	17000	90	LDEHAAN
...				
9001	ABBOTT	3000	10	abbott.m...

3

UPDATE
NEW_DEPTS

DEPARTMENT_ID	DEPARTMENT_NAME	TOT_DEPT_SA
10	Administration	940
20	Marketing	19000
30	Purchasing	30129
40	Human Resources	6500
...		



Differentiating Between Database Triggers and Stored Procedures

Triggers	Procedures
Defined with CREATE TRIGGER	Defined with CREATE PROCEDURE
Data dictionary contains source code in <code>USER_TRIGGERS</code>	Data dictionary contains source code in <code>USER_SOURCE</code>
Implicitly invoked	Explicitly invoked
COMMIT, SAVEPOINT, and ROLLBACK are not allowed	COMMIT, SAVEPOINT, and ROLLBACK are allowed



Managing Triggers

Disable or reenable a database trigger:

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

Disable or reenable all triggers for a table:

```
ALTER TABLE table_name DISABLE | ENABLE ALL TRIGGERS
```

Recompile a trigger for a table:

```
ALTER TRIGGER trigger_name COMPILE
```



DROP TRIGGER Syntax

To remove a trigger from the database, use the DROP TRIGGER syntax:

```
DROP TRIGGER trigger_name;
```

Example:

```
DROP TRIGGER secure_emp;
```

Trigger dropped.

Note: All triggers on a table are dropped when the table is dropped.



Trigger Test Cases

- **Test each triggering data operation, as well as nontriggering data operations.**
- **Test each case of the WHEN clause.**
- **Cause the trigger to fire directly from a basic data operation, as well as indirectly from a procedure.**
- **Test the effect of the trigger upon other triggers.**
- **Test the effect of other triggers upon the trigger.**



Trigger Execution Model and Constraint Checking

- 1. Execute all BEFORE STATEMENT triggers.**
- 2. Loop for each row affected:**
 - a. Execute all BEFORE ROW triggers.**
 - b. Execute all AFTER ROW triggers.**
- 3. Execute the DML statement and perform integrity constraint checking.**
- 4. Execute all AFTER STATEMENT triggers.**



Trigger Execution Model and Constraint Checking: Example

```
UPDATE employees SET department_id = 999
WHERE employee_id = 170;
-- Integrity constraint violation error
```

```
CREATE OR REPLACE TRIGGER constr_emp_trig
AFTER UPDATE ON employees
FOR EACH ROW
BEGIN
  INSERT INTO departments
    VALUES (999, 'dept999', 140, 2400);
END;
/
```

```
UPDATE employees SET department_id = 999
WHERE employee_id = 170;
-- Successful after trigger is fired
```



After Row and After Statement Triggers

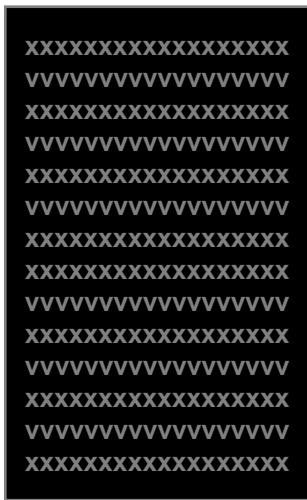
```
CREATE OR REPLACE TRIGGER audit_emp_trig
AFTER UPDATE or INSERT or DELETE on EMPLOYEES
FOR EACH ROW
BEGIN
    IF      DELETING      THEN  var_pack.set_g_del(1);
    ELSIF   INSERTING     THEN  var_pack.set_g_ins(1);
    ELSIF   UPDATING ('SALARY')
                      THEN  var_pack.set_g_up_sal(1);
    ELSE    var_pack.set_g_upd(1);
    END IF;
END audit_emp_trig;
/
```

```
CREATE OR REPLACE TRIGGER audit_emp_tab
AFTER UPDATE or INSERT or DELETE on employees
BEGIN
    audit_emp;
END audit_emp_tab;
/
```

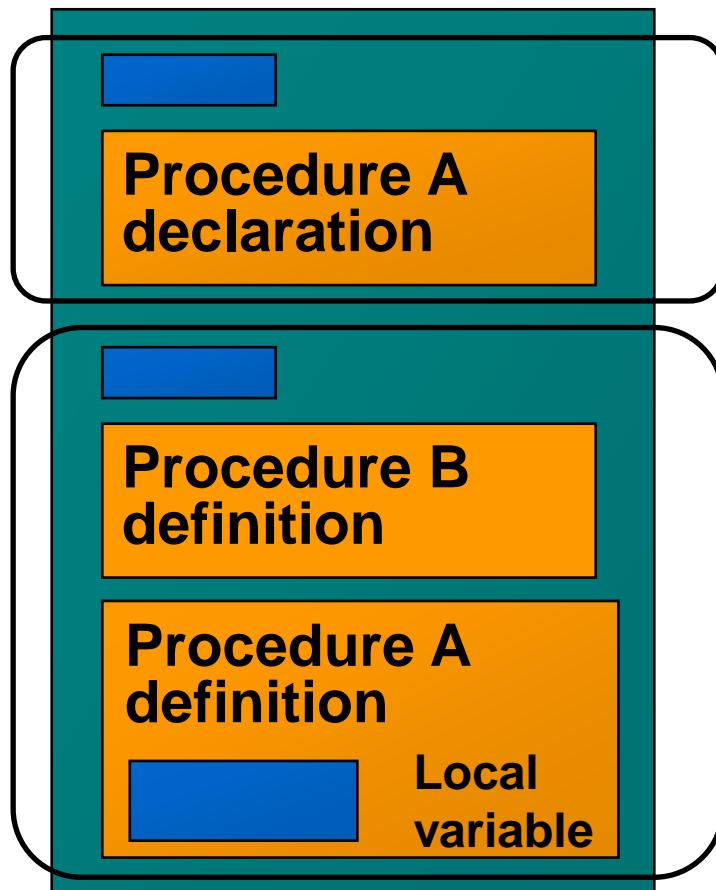


Summary

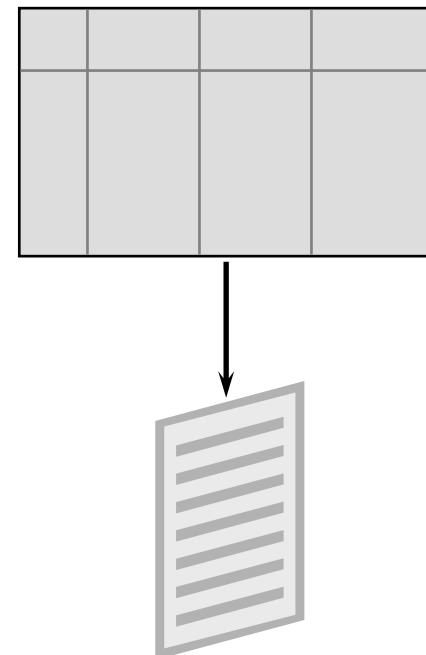
Procedure



Package



Trigger





More Trigger Concepts





Objectives

After completing this lesson, you should be able to do the following:

- Create additional database triggers
- Explain the rules governing triggers
- Implement triggers



Creating Database Triggers

- **Triggering user event:**
 - CREATE, ALTER, or DROP
 - Logging on or off
- **Triggering database or system event:**
 - Shutting down or starting up the database
 - A specific error (or any error) being raised



Creating Triggers on DDL Statements

Syntax:

```
CREATE [OR REPLACE] TRIGGER trigger_name
    timing
    [ddl_event1 [OR ddl_event2 OR ...]]
    ON {DATABASE | SCHEMA}
trigger_body
```



Creating Triggers on System Events

```
CREATE [OR REPLACE] TRIGGER trigger_name
  timing
  [database_event1 [OR database_event2 OR ...]]
  ON {DATABASE|SCHEMA}
trigger_body
```

LOGON and LOGOFF Trigger Example

```
CREATE OR REPLACE TRIGGER logon_trig
AFTER LOGON    ON    SCHEMA
BEGIN
  INSERT INTO log_trig_table(user_id, log_date, action)
  VALUES (USER, SYSDATE, 'Logging on');
END;
/
```

```
CREATE OR REPLACE TRIGGER logoff_trig
BEFORE LOGOFF   ON    SCHEMA
BEGIN
  INSERT INTO log_trig_table(user_id, log_date, action)
  VALUES (USER, SYSDATE, 'Logging off');
END;
/
```



CALL Statements

```
CREATE [OR REPLACE] TRIGGER trigger_name
timing
    event1 [OR event2 OR event3]
    ON table_name
    [REFERENCING OLD AS old | NEW AS new]
    [FOR EACH ROW]
    [WHEN condition]
    CALL procedure_name
```

```
CREATE OR REPLACE TRIGGER log_employee
BEFORE INSERT ON EMPLOYEES
    CALL log_execution
/
```

Reading Data from a Mutating Table

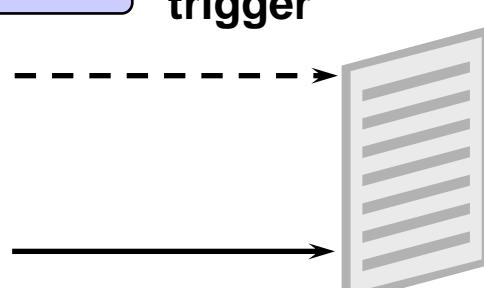
```
UPDATE employees
SET salary = 3400
WHERE last_name = 'Stiles';
```

EMPLOYEES table

EMPLOYEE_ID	LAST_NAME	JOB_ID	SALARY
125	Nayer	ST_CLERK	3200
126	Mikkilineni	ST_CLERK	2700
127	Landry	ST_CLERK	2400
...			
138	Stiles	ST_CLERK	3400
...			

Failure

CHECK_SALARY
trigger



BEFORE UPDATE row

Triggered table/
mutating table



Trigger event



Mutating Table: Example

```
CREATE OR REPLACE TRIGGER check_salary
    BEFORE INSERT OR UPDATE OF salary, job_id
    ON employees
    FOR EACH ROW
    WHEN (NEW.job_id <> 'AD_PRES')
DECLARE
    v_minsalary employees.salary%TYPE;
    v_maxsalary employees.salary%TYPE;
BEGIN
    SELECT MIN(salary), MAX(salary)
        INTO v_minsalary, v_maxsalary
        FROM employees
        WHERE job_id = :NEW.job_id;
    IF :NEW.salary < v_minsalary OR
        :NEW.salary > v_maxsalary THEN
        RAISE_APPLICATION_ERROR(-20505,'Out of range');
    END IF;
END;
/
```



Mutating Table: Example

```
UPDATE employees
    SET salary = 3400
 WHERE last_name = 'Stiles';
```

```
UPDATE employees
    *
```

ERROR at line 1:

ORA-04091: table PLSQL.EMPLOYEES is mutating, trigger/function may not see it

ORA-06512: at "PLSQL.CHECK_SALARY", line 5

ORA-04088: error during execution of trigger 'PLSQL.CHECK_SALARY'



Benefits of Database Triggers

- **Improved data security:**
 - Provide enhanced and complex security checks
 - Provide enhanced and complex auditing
- **Improved data integrity:**
 - Enforce dynamic data integrity constraints
 - Enforce complex referential integrity constraints
 - Ensure that related operations are performed together implicitly



Managing Triggers

The following system privileges are required to manage triggers:

- **The CREATE/ALTER/DROP (ANY) TRIGGER privilege enables you to create a trigger in any schema**
- **The ADMINISTER DATABASE TRIGGER privilege enables you to create a trigger on DATABASE**
- **The EXECUTE privilege (if your trigger refers to any objects that are not in your schema)**

Note: Statements in the trigger body operate under the privilege of the trigger owner, not the trigger user.



Viewing Trigger Information

You can view the following trigger information:

- **USER_OBJECTS** data dictionary view: object information
- **USER_TRIGGERS** data dictionary view: the text of the trigger
- **USER_ERRORS** data dictionary view: PL/SQL syntax errors (compilation errors) of the trigger



Using USER_TRIGGER*

Column	Column Description
TRIGGER_NAME	Name of the trigger
TRIGGER_TYPE	The type is BEFORE, AFTER, INSTEAD OF
TRIGGERING_EVENT	The DML operation firing the trigger
TABLE_NAME	Name of the database table
REFERENCING_NAMES	Name used for :OLD and :NEW
WHEN_CLAUSE	The when_clause used
STATUS	The status of the trigger
TRIGGER_BODY	The action to take

* Abridged column list



Listing the Code of Triggers

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

trigger_name	trigger_type	triggering_event	table_name	referencing_names	when_clause	status	trigger_body
RESTRICT_SALARY	BEFORE EACH ROW	INSERT OR UPDATE	EMPLOYEES	REFERENCING NEW AS NEW OLD AS OLD		ENABLED	BEGIN IF NOT (:NEW.JOB_ID IN ('AD_PRÉS', 'AD_VP')) AND :NEW.SAL



Summary

In this lesson, you should have learned how to:

- Use advanced database triggers
- List mutating and constraining rules for triggers
- Describe the real-world application of triggers
- Manage triggers
- View trigger information



Collections

- A collection is a group of elements of homogenous data types. It generally comprises arrays, lists, sets, and so on.
- Each of the elements has a particular subscript which reflects its position.
- PL SQL Collections are of the following types:
- Associative Array/Index-by tables
- Nested tables.
- Varrays.
- PL SQL collections are generally used for storage and manipulation of big chunks of data, using the keyword BULK COLLECT in Oracle.



Oracle Objects

- Oracle object technology is a layer of abstraction built on Oracle relational technology.
- New object types can be created from any built-in database types and any previously created object types, object references, and collection types.
- Metadata for user-defined types is stored in a schema that is available to SQL, PL/SQL, Java, and other published interfaces.



Oracle Objects

- Object types and related object-oriented features such as variable-length arrays and nested tables provide higher-level ways to organize and access data in the database.
- Underneath the object layer, data is still stored in columns and tables, but you are able to work with the data in terms of the real-world entities, such as customers and purchase orders, that make the data meaningful.
- Instead of thinking in terms of columns and tables when you query the database, you can simply select a customer.



Advantages of Objects

- In general, the object-type model is similar to the class mechanism found in C++ and Java.
- Like classes, objects make it easier to model complex, real-world business entities and logic, and the reusability of objects makes it possible to develop database applications faster and more efficiently.
- By natively supporting object types in the database, Oracle enables application developers to directly access the data structures used by their applications.



Advantages of Objects

- No mapping layer is required between client-side objects and the relational database columns and tables that contain the data.
- Object abstraction and the encapsulation of object behaviors also make applications easier to understand and maintain.



Objects Can Encapsulate Operations Along with Data

- **Database tables contain only data.**
- **Objects can include the ability to perform operations that are likely to be needed on that data.**
- **Thus a purchase order object might include a method to sum the cost of all the items purchased.**
- **Or a customer object might have methods to return the customer's buying history and payment pattern.**
- **An application can simply call the methods to retrieve the information.**



Objects Are Efficient

- Object types and their methods are stored with the data in the database, so they are available for any application to use.
- Developers can benefit from work that is already done and do not need to re-create similar structures in every application.
- You can fetch and manipulate a set of related objects as a single unit.
- A single request to fetch an object from the server can retrieve other objects that are connected to it.



Objects Are Efficient

- For example, when you select a customer object and get the customer's name, phone, and the multiple parts of his address in a single round-trip between the client and the server.
- When you reference a column of a SQL object type, you retrieve the whole object.



Key Features of the Object-Relational Model

Object Type <i>person_typ</i>	
Attributes	Methods
idno	get_idno
first_name	display_details
last_name	
email	
phone	

Object
idno: 65
first_name: Verna
last_name: Mills
email: vmills@oracle.com
phone: 1-800-555-4412

Object
idno: 101
first_name: John
last_name: Smith
email: jsmith@oracle.com
phone: 1-800-555-1212



Object Types

- Define A TYPE
- Define A TYPE BODY
- Defining A Table
- Constructors
- Data Access
- Issues



UTL_FILE

- **conn / as sysdba**
- **grant execute on utl_file to system;**



PLSQL UTL_File

- Oracle 7.3 introduced the UTL_FILE package.
- The UTL_FILE package in the Oracle database is used to write data from the Oracle database to the OS file and also read from the OS file to the Oracle database.
- Mainly three procedures are given in the UTL_File package to read or write the data.
- These procedures are put_line(), putf(), and get_line().



PLSQL UTL_File

- The `put_line()` and `putf()` procedures are used to write data into an OS file.
- But the `get_line()` procedure of the `UTL_FILE` package is used to read data from an OS file to the Oracle database.



Pre-requisite to Use UTL_FILE Package in Oracle

- To write UTL_FILE programs, we must create an alias directory related to the physical directory.
- Syntax to create alias directory related to a physical directory.
- Syntax:-
- **CREATE OR REPLACE DIRECTORY dirname AS 'path';**



Pre-requisite to Use UTL_FILE Package in Oracle

- But to create alias directory user must have '**'CREATE ANY DIRECTORY'** system privileges, otherwise, the Oracle server returns an error.
- The database administrator gives '**'CREATE ANY DIRECTORY'** system privileges to the user by the following syntax.
- **Syntax:-**
- **GRANT CREATE ANY DIRECTORY TO username;**



Pre-requisite to Use UTL_FILE Package in Oracle

- Before performing read, write operations at alias directory the user must have read and write object privileges on that alias directory.
- The Oracle database administrator must give read, write object privileges on alias directory by using the following syntax.
- Syntax:-
- GRANT READ, WRITE ON DIRECTORY
directoryname TO username;



Pre-requisite to Use UTL_FILE Package in Oracle

- **Writing or Storing Data Into an OS File**
- **We have to follow four steps to write or store data into an OS file.**
- **These four steps are the following:-**
- **1) Declare file pointer variable**
- **2) Open the file by using FOPEN() function**
- **3) Write data into the file using PUTF() or PUT_LINE() procedure**
- **4) Close the file using FCLOSE() function**



Pre-requisite to Use UTL_FILE Package in Oracle

- **Step1:- Before we are opening a file, we must declare the file pointer variable in the declare section of the PL SQL block by using FILE_TYPE from the UTL_FILE package.**
- **Syntax:-**
- **Filepointervarname UTL_FILE.FILE_TYPE;**



Pre-requisite to Use UTL_FILE Package in Oracle

- **Step2:- Before we are writing data into an OS file, then we must open the file by using FOPEN() function from the utl_file package. This function is used in the executable section of the PL SQL block. The FOPEN() function accepts three parameters and returns the file_type data type.**
- **Syntax:-**
- **filePointerVarname :=
UTL_FILE.FOPEN('aliasdirectoryname', 'filename',
'mode');**



Pre-requisite to Use UTL_FILE Package in Oracle

- Here aliasDirectoryName must be in capital letter.

- Different modes used in fopen() are,
 - • w:- write mode
 - • r:- read mode
 - • a:- append mode



Pre-requisite to Use UTL_FILE Package in Oracle

- Step3:- To write data into the OS file we can use **put_line()** or **putf()** procedure from **UTL_FILE package**.
- Syntax for **putf()**:-
UTL_FILE.PUT(filepointervarname, 'content');
- Syntax for **put_line()**:-
UTL_FILE.PUT_LINE(filepointervarname, format);
- Step4:- At last, we must close the file by using **fclose()** procedure from **utl_file package**.
- Syntax:-
UTL_FILE.FCLOSE(filepointervarname);

The data types for large objects that were introduced in Oracle Database 8

Type	Description
CLOB	Character large object. Stores up to 8 terabytes of character data inside the database.
NCLOB	National character large object. Stores up to 8 terabytes of national character data inside the database.
BLOB	Binary large object. Stores up to 8 terabytes of binary data inside the database.
BFILE	Binary file. Stores a pointer to a large binary file stored outside the database in the file system of the host computer.

CLOB Character large object. Stores up to 8 terabytes of character data inside the database.

NCLOB National character large object. Stores up to 8 terabytes of national character data inside the database.

BLOB Binary large object. Stores up to 8 terabytes of binary data inside the database.

BFILE Binary file. Stores a pointer to a large binary file stored outside the database in the file system of the host computer.



Managing Dependencies





Objectives

After completing this lesson, you should be able to do the following:

- Track procedural dependencies
- Predict the effect of changing a database object upon stored procedures and functions
- Manage procedural dependencies

Understanding Dependencies

Dependent Objects

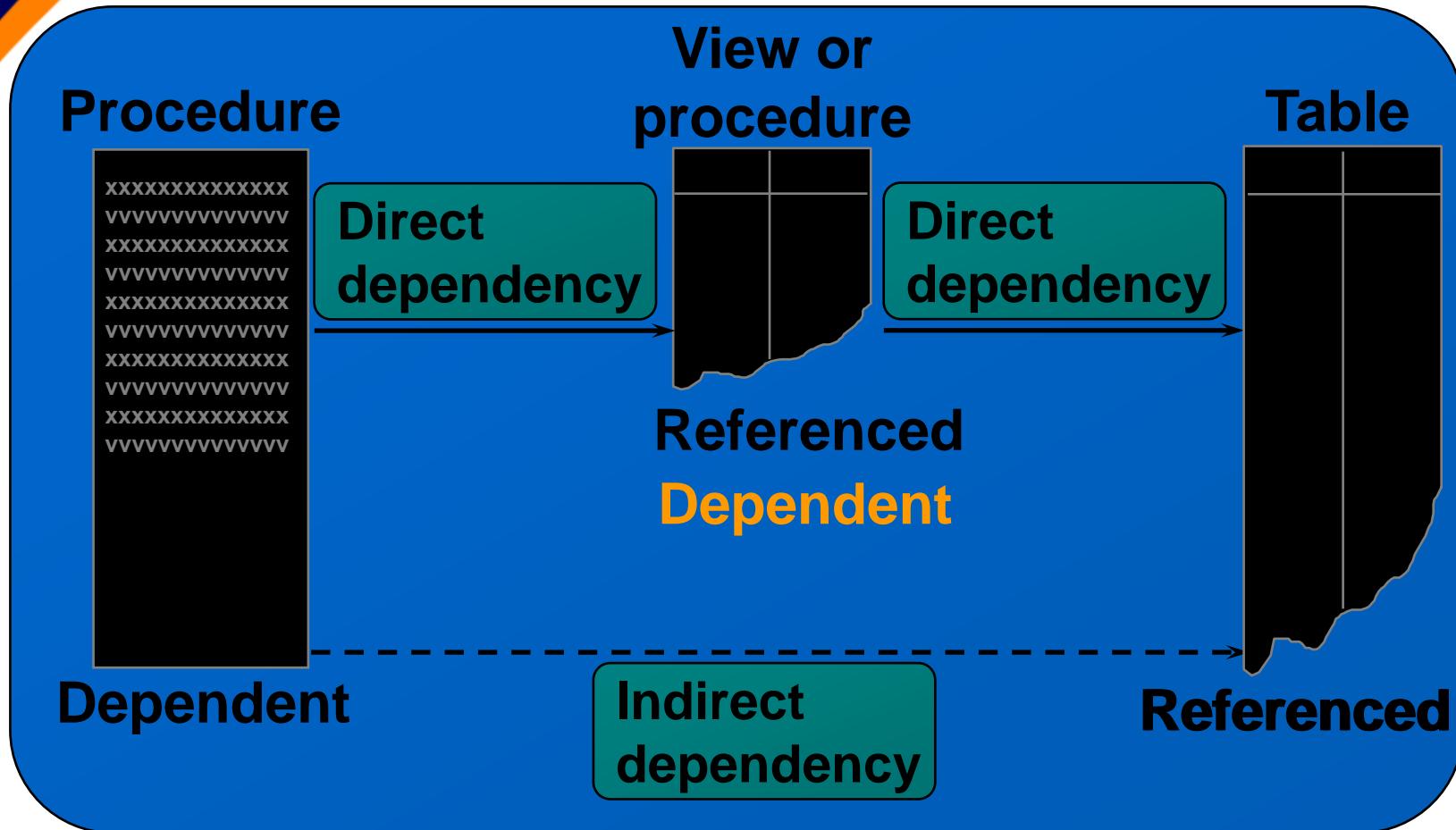
Table
View
Database Trigger
Procedure
Function
Package Body
Package Specification
User-Defined Object
and Collection Types

Referenced Objects

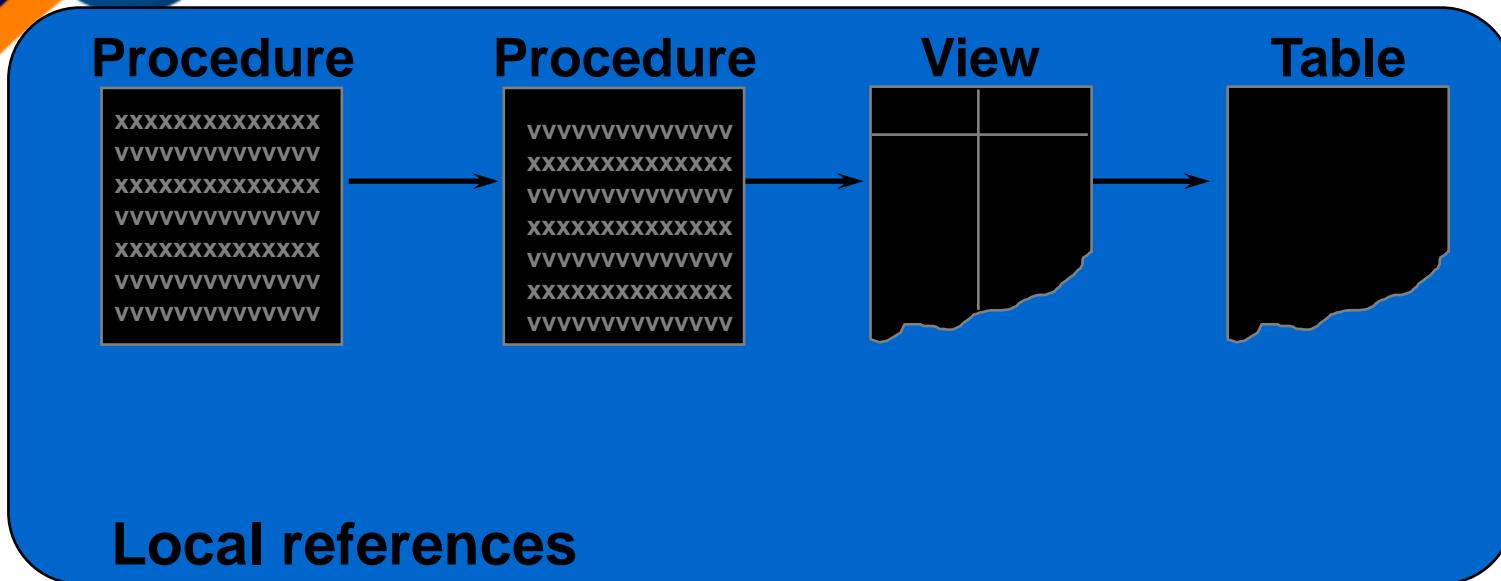
Function
Package Specification
Procedure
Sequence
Synonym
Table
View
User-Defined Object
and Collection Types



Dependencies

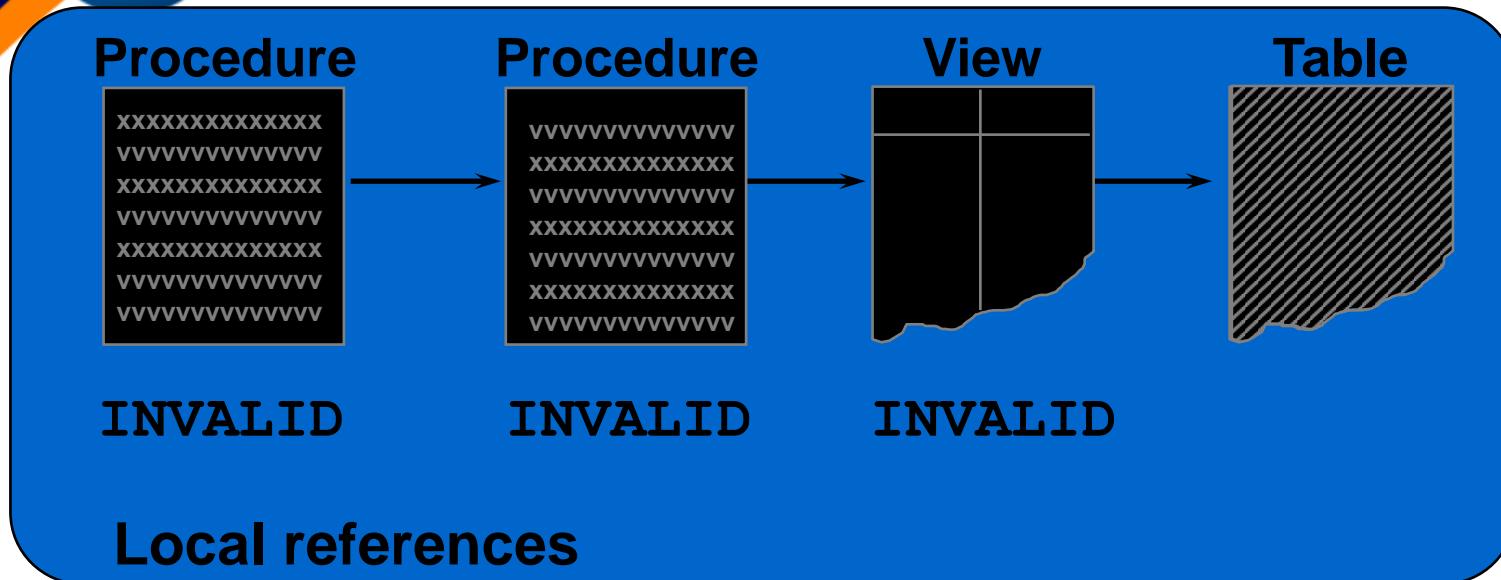


Local Dependencies



Direct local dependency

Local Dependencies



→
Direct local dependency


Definition change

The Oracle server implicitly recompiles any INVALID object when the object is next called.

A Scenario of Local Dependencies

**ADD_EMP
procedure**



EMP_VW view

EMPLOYEE_ID	LAST_NAME	FIRST_NAME	EMAIL	DEPARTMENT
100	King	Steven	SKING	
101	Kochhar	Neena	NKOCHHAR	
102	De Haan	Lex	LDEHAAN	
105	Austin	David	DAUSTIN	
108	Greenberg	Nancy	NGREENBERG	

...

**QUERY_EMP
procedure**



EMPLOYEES table

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER
100	Steven	King	SKING	515.123.4561
101	Neena	Kochhar	NKOCHHAR	515.123.4562
102	Lex	De Haan	LDEHAAN	515.123.4563
105	David	Austin	DAUSTIN	590.423.4564
108	Nancy	Greenberg	NGREENBERG	515.124.4565

...



Displaying Direct Dependencies by Using USER_DEPENDENCIES

```
SELECT name, type, referenced_name, referenced_type  
FROM user_dependencies  
WHERE referenced_name IN ('EMPLOYEES', 'EMP_VW' );
```

NAME	TYPE	REFERENCED_NAME	REFERENCED_T
EMP_DETAILS_VIEW	VIEW	EMPLOYEES	TABLE
...			
EMP_VW	VIEW	EMPLOYEES	TABLE
...			
QUERY_EMP	PROCEDURE	EMPLOYEES	TABLE
ADD_EMP	PROCEDURE	EMP_VW	VIEW



Displaying Direct and Indirect Dependencies

1. Run the script `utldtree.sql` that creates the objects that enable you to display the direct and indirect dependencies.
2. Execute the `DEPTREE_FILL` procedure.

```
EXECUTE deptree_fill('TABLE', 'SCOTT', 'EMPLOYEES')
```

PL/SQL procedure successfully completed.

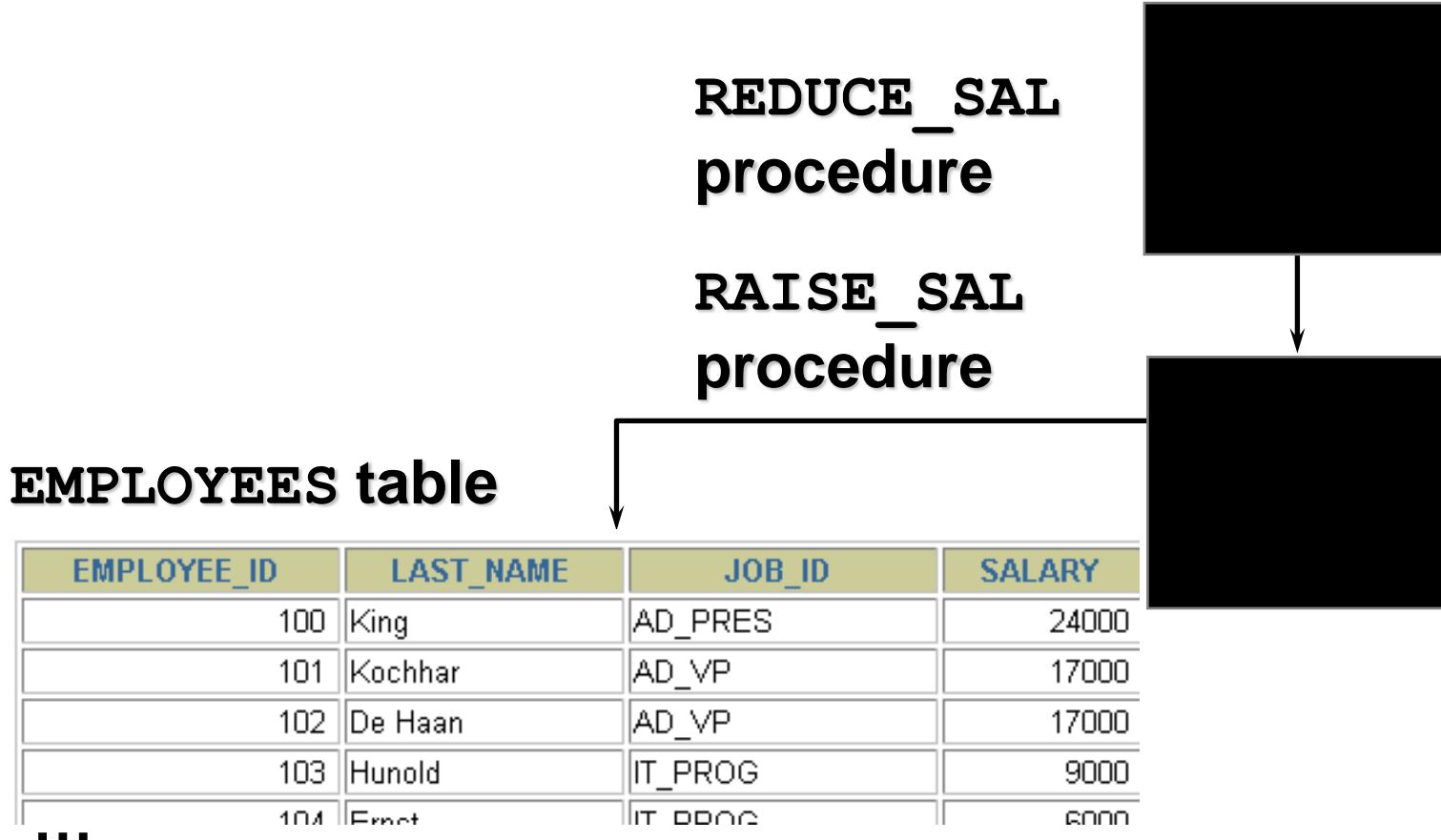
Displaying Dependencies

DEPTREE View

```
SELECT      nested_level, type, name  
FROM        deptree  
ORDER BY    seq#;
```

NESTED_LEVEL	TYPE	NAME
0	TABLE	EMPLOYEES
1	VIEW	EMP_DETAILS_VIEW
...		
1	TRIGGER	CHECK_SALARY
1	VIEW	EMP_VW
2	PROCEDURE	ADD_EMP
1	PACKAGE	MGR_CONSTRAINTS_PKG
2	TRIGGER	CHECK_PRES_TITLE
...		

Another Scenario of Local Dependencies



A Scenario of Local Naming Dependencies

QUERY_EMP
procedure



EMPLOYEES
table

EMPLOYEES public synonym

EMPLOYEE_ID	LAST_NAME	JOB_ID	SALARY
100	King	AD_PRES	24000
101	Kochhar	AD_VP	17000
102	De Haan	AD_VP	17000
103	Hunold	IT_PROG	9000
104	Ernst	IT_PDG	6000
...			

EMPLOYEE_ID	LAST_NAME	JOB_ID	SALARY
100	King	AD_PRES	24000
101	Kochhar	AD_VP	17000
102	De Haan	AD_VP	17000
103	Hunold	IT_PROG	9000
104	Ernst	IT_PDG	6000
...			

Understanding Remote Dependencies

Procedure

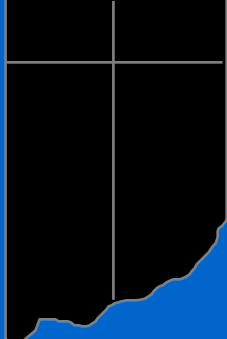
```
xxxxxxxxxxxxxx  
vvvvvvvvvvvvv  
xxxxxxxxxxxxxx  
vvvvvvvvvvvvv
```

Network

Procedure

```
vvvvvvvvvvvvv  
xxxxxxxxxxxxxx  
vvvvvvvvvvvvv
```

View



Table



Local and remote references



Direct local
dependency



Direct remote
dependency

Understanding Remote Dependencies

Procedure

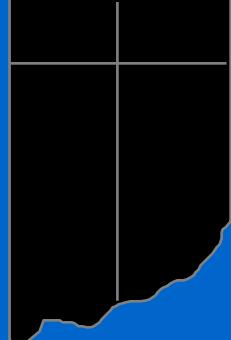
```
xxxxxxxxxxxxxx  
vvvvvvvvvvvvv  
xxxxxxxxxxxxxx  
vvvvvvvvvvvvv
```

Network

Procedure

```
vvvvvvvvvvvvv  
xxxxxxxxxxxxxx  
vvvvvvvvvvvvv  
xxxxxxxxxxxxxx  
vvvvvvvvvvvvv  
xxxxxxxxxxxxxx  
vvvvvvvvvvvvv
```

View



Table



VALID

INVALID

INVALID

Local and remote references



Direct local
dependency



Direct remote
dependency



Definition
change



Concepts of Remote Dependencies

Remote dependencies are governed by the mode chosen by the user:

- **TIMESTAMP** checking
- **SIGNATURE** checking



REMOTE_DEPENDENCIES_MODE Parameter

Setting REMOTE_DEPENDENCIES_MODE:

- As an `init.ora` parameter

`REMOTE_DEPENDENCIES_MODE = value`

- At the system level

`ALTER SYSTEM SET`

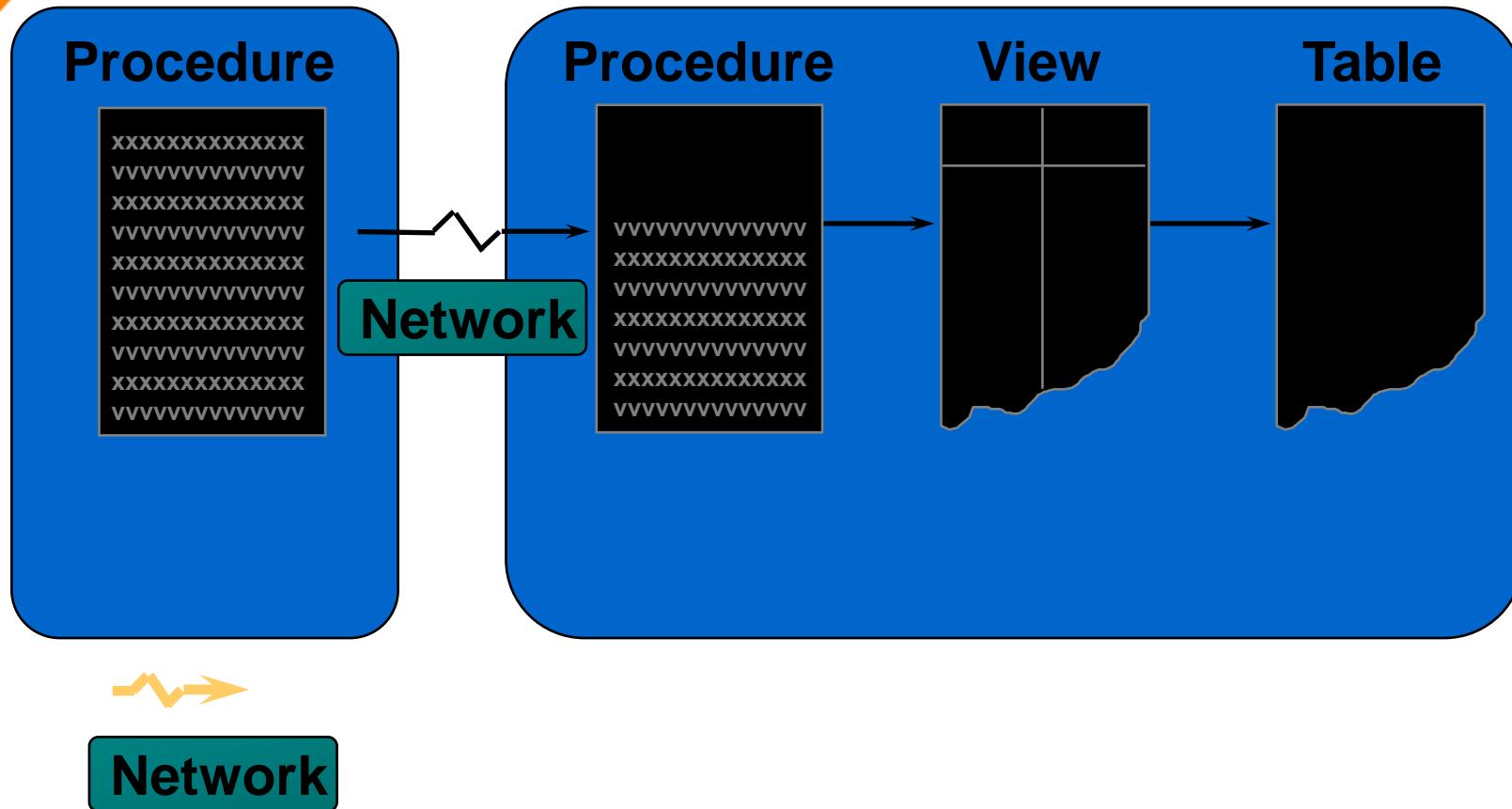
`REMOTE_DEPENDENCIES_MODE = value`

- At the session level

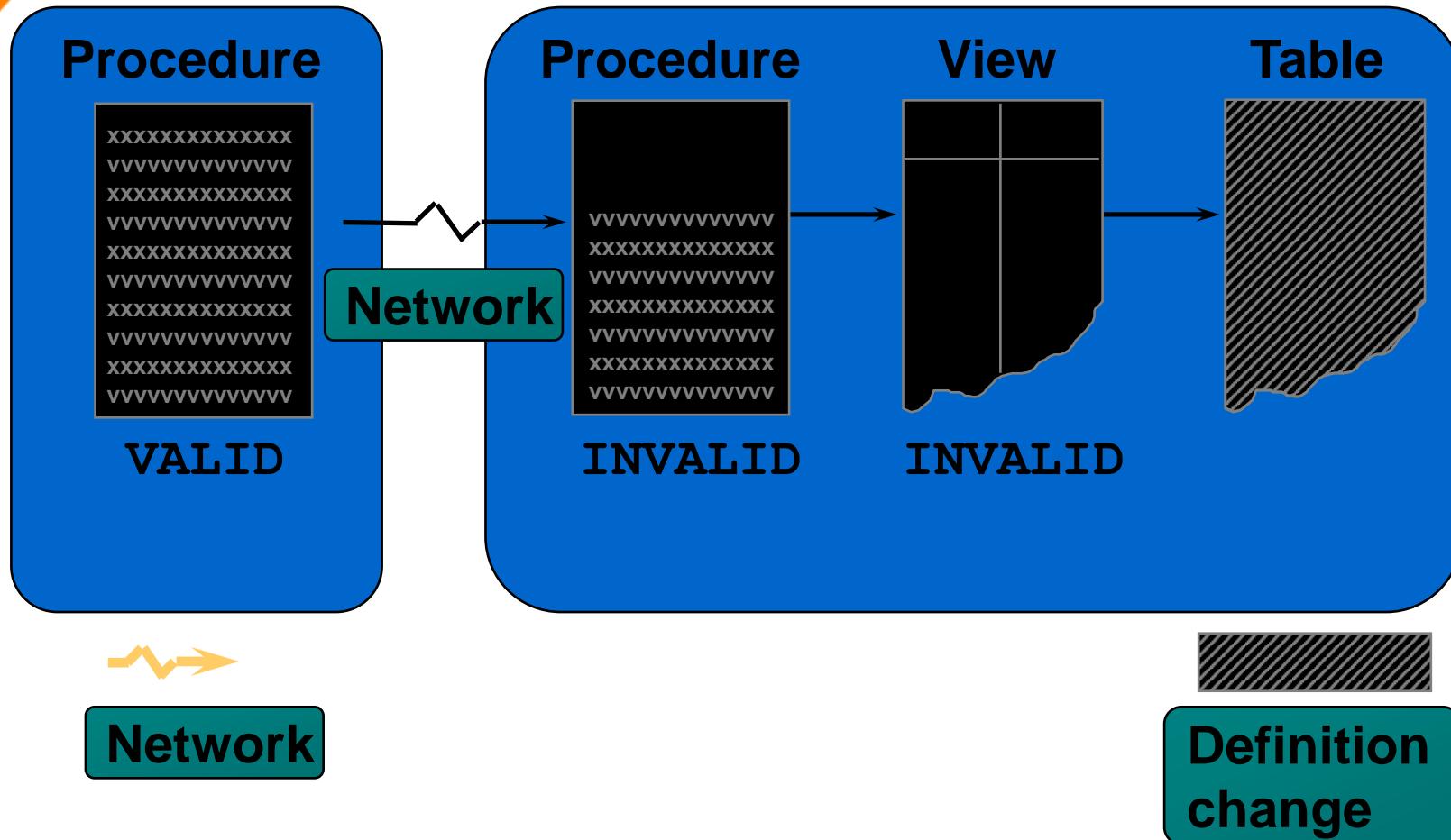
`ALTER SESSION SET`

`REMOTE_DEPENDENCIES_MODE = value`

Remote Dependencies and Time Stamp Mode



Remote Dependencies and Time Stamp Mode





Signature Mode

- **The signature of a procedure is:**
 - The name of the procedure
 - The datatypes of the parameters
 - The modes of the parameters
- **The signature of the remote procedure is saved in the local procedure.**
- **When executing a dependent procedure, the signature of the referenced remote procedure is compared.**



Recompiling a PL/SQL Program Unit

Recompilation:

- Is handled automatically through implicit run-time recompilation
- Is handled through explicit recompilation with the ALTER statement

```
ALTER PROCEDURE [SCHEMA.]procedure_name COMPILE;
```

```
ALTER FUNCTION [SCHEMA.]function_name COMPILE;
```

```
ALTER PACKAGE [SCHEMA.]package_name COMPILE [PACKAGE];  
ALTER PACKAGE [SCHEMA.]package_name COMPILE BODY;
```

```
ALTER TRIGGER trigger_name [COMPILE [DEBUG]] ;
```



Unsuccessful Recompilation

Recompiling dependent procedures and functions is unsuccessful when:

- **The referenced object is dropped or renamed**
- **The data type of the referenced column is changed**
- **The referenced column is dropped**
- **A referenced view is replaced by a view with different columns**
- **The parameter list of a referenced procedure is modified**



Successful Recompilation

Recompiling dependent procedures and functions is successful if:

- **The referenced table has new columns**
- **The data type of referenced columns has not changed**
- **A private table is dropped, but a public table, having the same name and structure, exists**
- **The PL/SQL body of a referenced procedure has been modified and recompiled successfully**

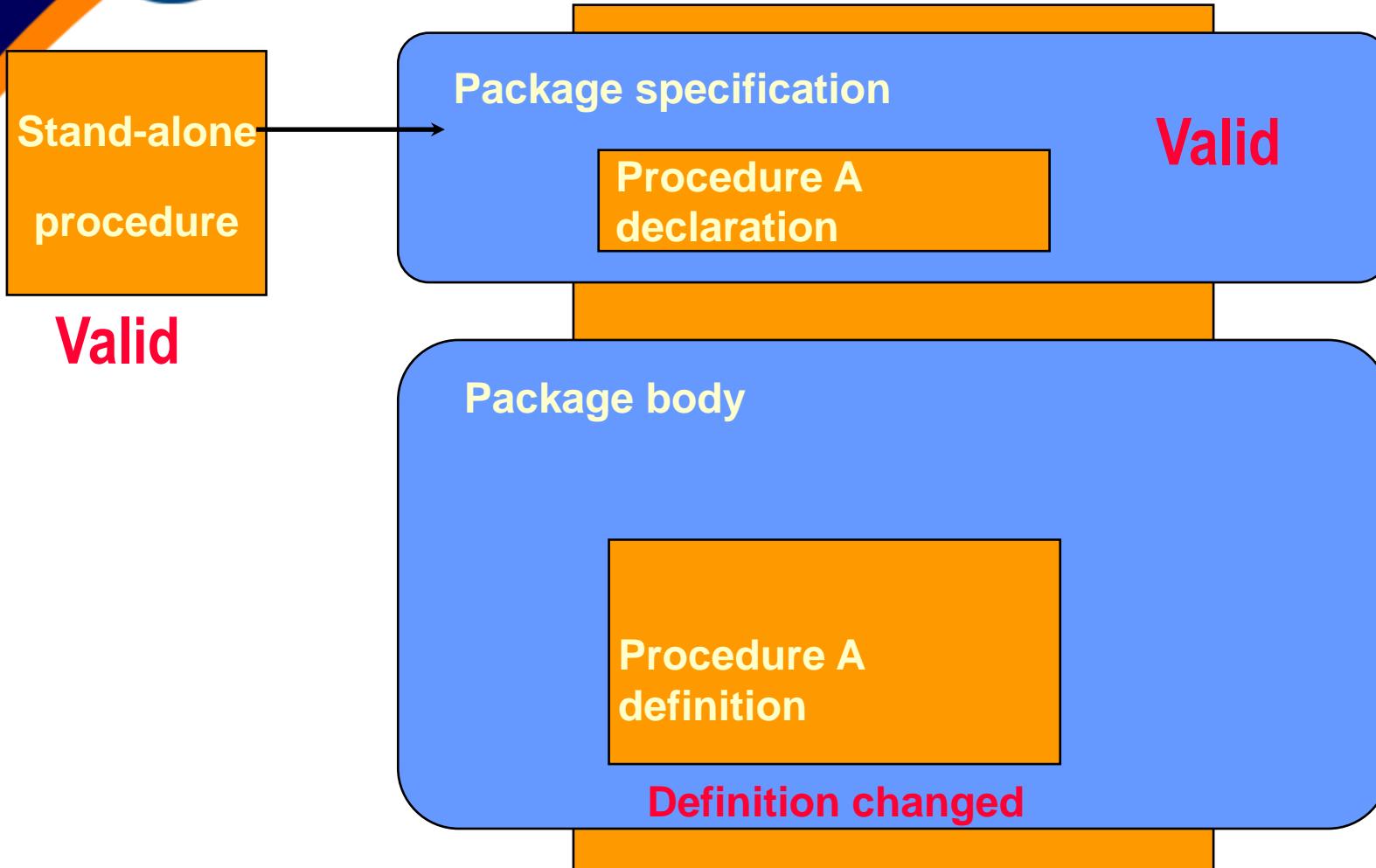


Recompilation of Procedures

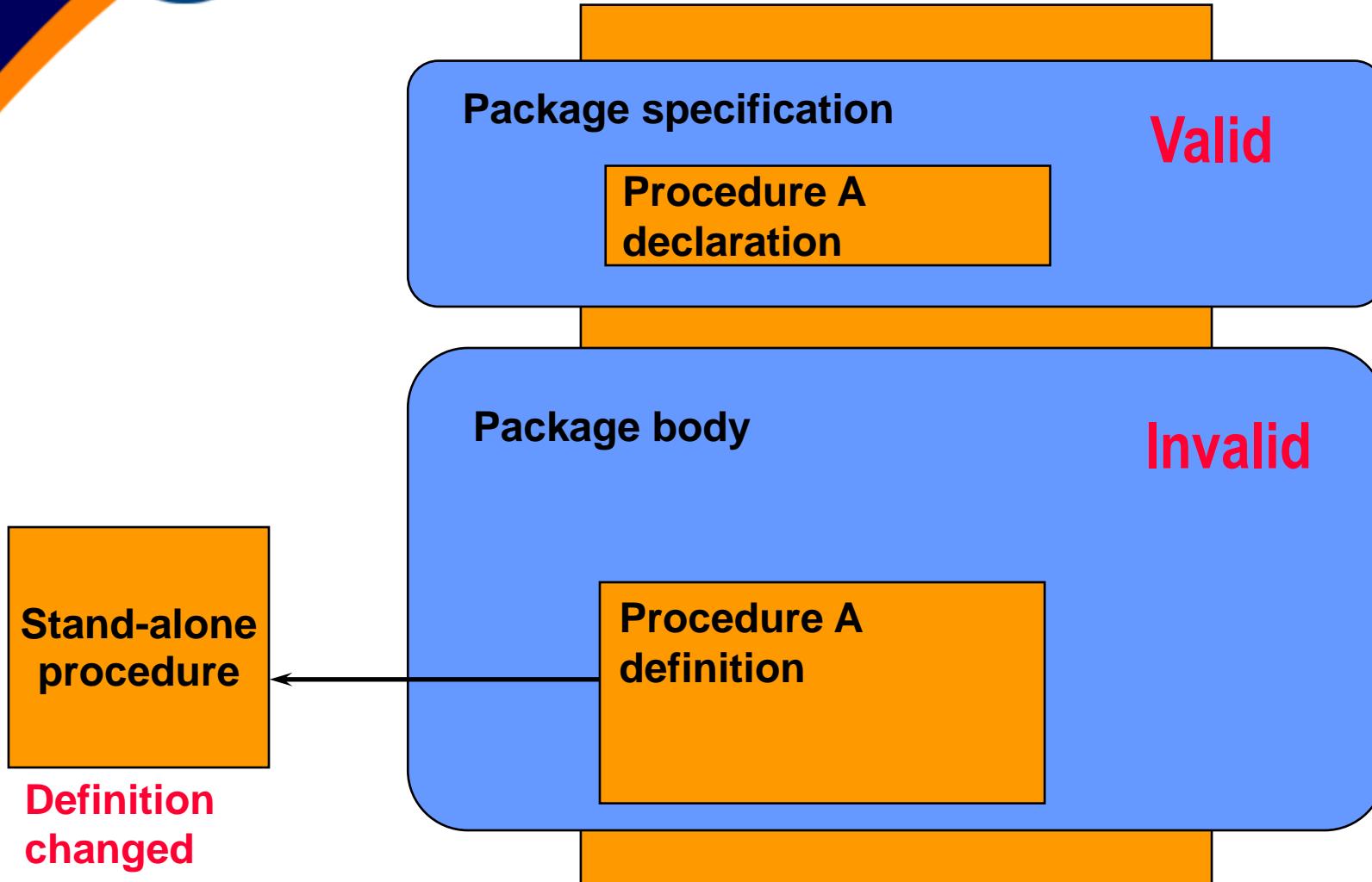
Minimize dependency failures by:

- Declaring records by using the %ROWTYPE attribute
- Declaring variables with the %TYPE attribute
- Querying with the SELECT * notation
- Including a column list with INSERT statements

Packages and Dependencies



Packages and Dependencies





Summary

In this lesson, you should have learned how to:

- **Keep track of dependent procedures**
- **Recompile procedures manually as soon as possible after the definition of a database object changes**

We
Are IT



Thank You



info@ness.com