Oracle Database 10*g*: Develop PL/SQL Program Units

Electronic Presentation

D17169GC21
Production 2.1
December 2006
D48233

ORACLE

Authors

Tulika Srivastava
Glenn Stokol

Technical Contributors and Reviewers

Chaitanya Koratamaddi
Dr. Christoph Burandt
Zarko Cesljas
Yanti Chang
Kathryn Cunningham
Burt Demchick
Laurent Dereac
Peter Driver
Bryan Roberts
Bryn Llewellyn
Nancy Greenberg
Craig Hollister
Thomas Hoogerwerf
Taj-Ul Islam
Inger Joergensen
Eric Lee
Malika Marghadi
Hildegard Mayr
Nagavalli Pataballa
Sunitha Patel
Srinivas Putrevu
Denis Raphaely
Helen Robertson
Grant Spencer
Glenn Stokol
Tone Thomas
Priya Vennapusa
Lex Van Der Werff

Graphic Designer

Satish Bettegowda

Editors

Nita Pavitran
Richard Wallis

Publisher

Sheryl Domingue

Introduction

Lesson Objectives

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

- **Discuss the goals of the course**
- **Identify the modular components of PL/SQL:**
  - **Anonymous blocks**
  - **Procedures and functions**
  - **Packages**
- **Discuss the PL/SQL execution environment**
- **Describe the database schema and tables that are used in the course**
- **List the PL/SQL development environments that are available in the course**

ORACLE

Course Objectives

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

- Create, execute, and maintain:
  - Procedures and functions with `OUT` parameters
  - Package constructs
  - Database triggers
- Manage PL/SQL subprograms and triggers
- Use a subset of Oracle-supplied packages to:
  - Generate screen, file, and Web output
  - Schedule PL/SQL jobs to run independently
- Build and execute dynamic SQL statements
- Manipulate large objects (LOBs)

Course Agenda

Lessons for day 1:

I. **Introduction**
1. **Creating Stored Procedures**
2. **Creating Stored Functions**
3. **Creating Packages**
4. **Using More Package Concepts**

Course Agenda

Lessons for day 2:

5. **Using Oracle-Supplied Packages in Application Development**
6. **Dynamic SQL and Metadata**
7. **Design Considerations for PL/SQL Code**
8. **Managing Dependencies**

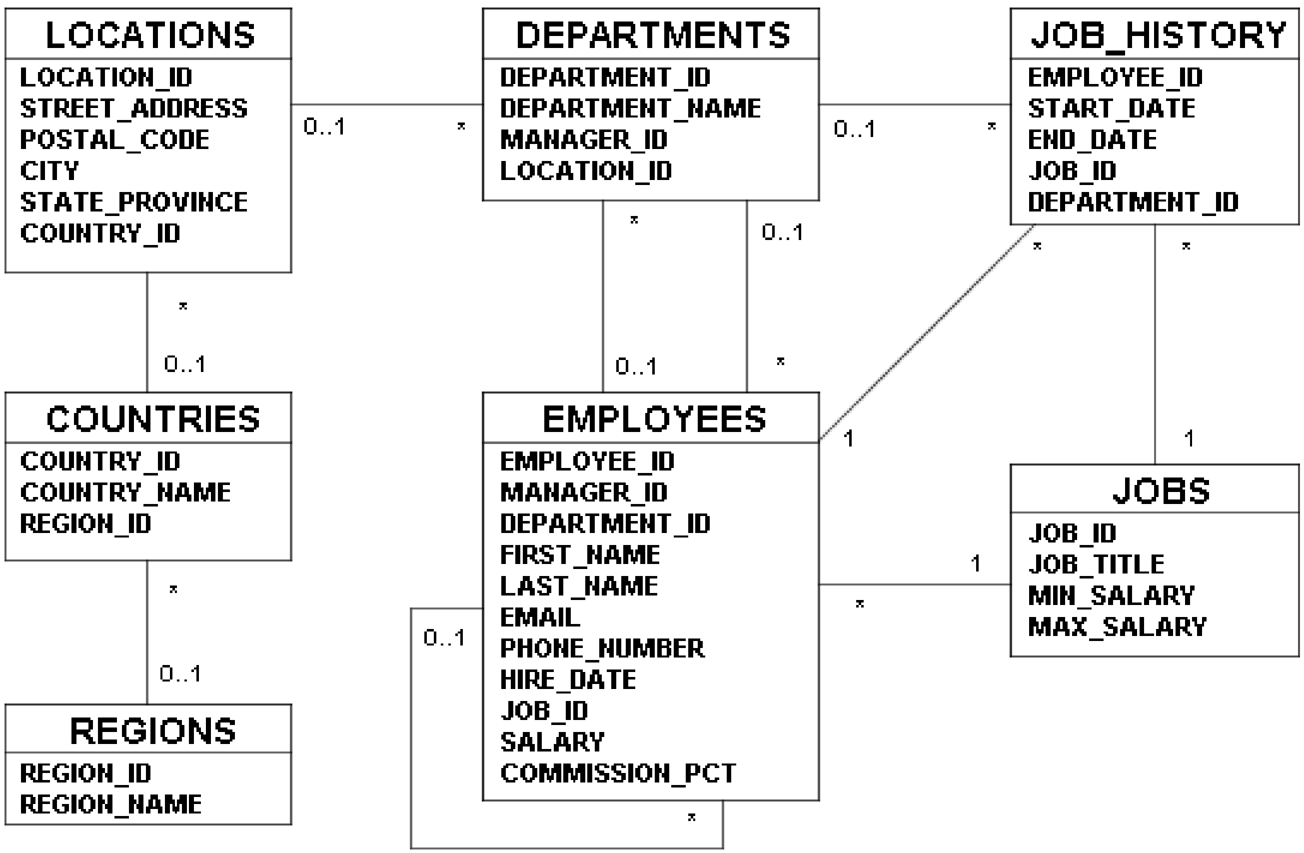
Course Agenda

Lessons for day 3:

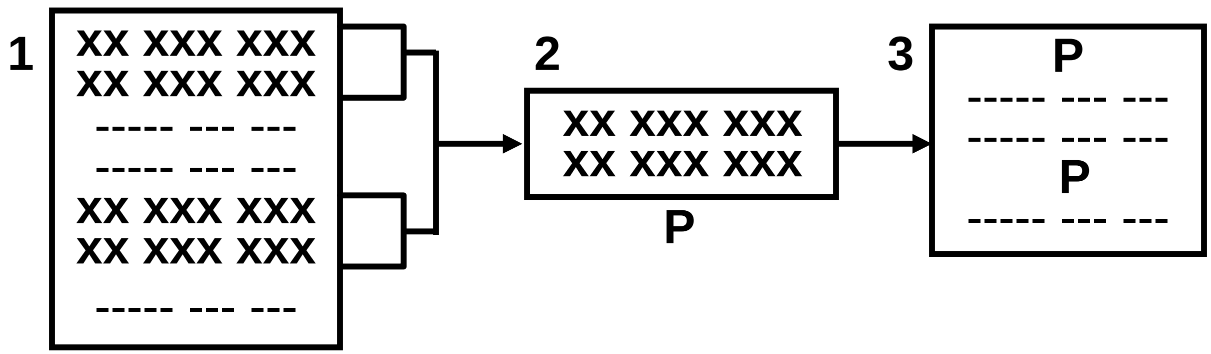
9. Manipulating Large Objects
10. Creating Triggers
11. Applications for Triggers
12. Understanding and Influencing the PL/SQL Compiler

ORACLE

Human Resources (HR) Schema



Creating a Modularized and Layered Subprogram Design



- **Modularize code into subprograms.**
  1. **Locate code sequences repeated more than once.**
  2. **Create subprogram P containing the repeated code.**
  3. **Modify original code to invoke the new subprogram.**
- **Create subprogram layers for your application.**
  - **Data access subprogram layer with SQL logic**
  - **Business logic subprogram layer, which may or may not use data access layer**

Modularizing Development with PL/SQL Blocks

- PL/SQL is a block-structured language. The PL/SQL code block helps modularize code by using:
  - Anonymous blocks
  - Procedures and functions
  - Packages
  - Database triggers
- The benefits of using modular program constructs are:
  - Easy maintenance
  - Improved data security and integrity
  - Improved performance
  - Improved code clarity

Review of Anonymous Blocks

Anonymous blocks:

- **Form the basic PL/SQL block structure**
- **Initiate PL/SQL processing tasks from applications**
- **Can be nested within the executable section of any PL/SQL block**

```
[DECLARE       -- Declaration Section (Optional)
  variable declarations; ... ]
BEGIN          -- Executable Section (Mandatory)
  SQL or PL/SQL statements;
[EXCEPTION     -- Exception Section (Optional)
  WHEN exception THEN statements; ]
END;           -- End of Block (Mandatory)
```

ORACLE

Introduction to PL/SQL Procedures

Procedures are named PL/SQL blocks that perform a sequence of actions.

```
CREATE PROCEDURE getemp IS  -- header
  emp_id  employees.employee_id%type;
  lname   employees.last_name%type;
BEGIN
  emp_id := 100;
  SELECT last_name INTO lname
  FROM EMPLOYEES
  WHERE employee_id = emp_id;
  DBMS_OUTPUT.PUT_LINE('Last name: '||lname);
END;
/
```

Introduction to PL/SQL Functions

Functions are named PL/SQL blocks that perform a sequence of actions and return a value. A function can be invoked from:

- Any PL/SQL block
- A SQL statement (subject to some restrictions)

```
CREATE FUNCTION avg_salary RETURN NUMBER IS
  avg_sal employees.salary%type;
BEGIN
  SELECT AVG(salary) INTO avg_sal
  FROM EMPLOYEES;
  RETURN avg_sal;
END;
/
```

Introduction to PL/SQL Packages

PL/SQL packages have a specification and an optional body. Packages group related subprograms together.

```
CREATE PACKAGE emp_pkg IS
  PROCEDURE getemp;
  FUNCTION avg_salary RETURN NUMBER;
END emp_pkg;
/
CREATE PACKAGE BODY emp_pkg IS
  PROCEDURE getemp IS ...
  BEGIN ... END;

  FUNCTION avg_salary RETURN NUMBER IS ...
  BEGIN ... RETURN avg_sal; END;
END emp_pkg;
/
```

Introduction to PL/SQL Triggers

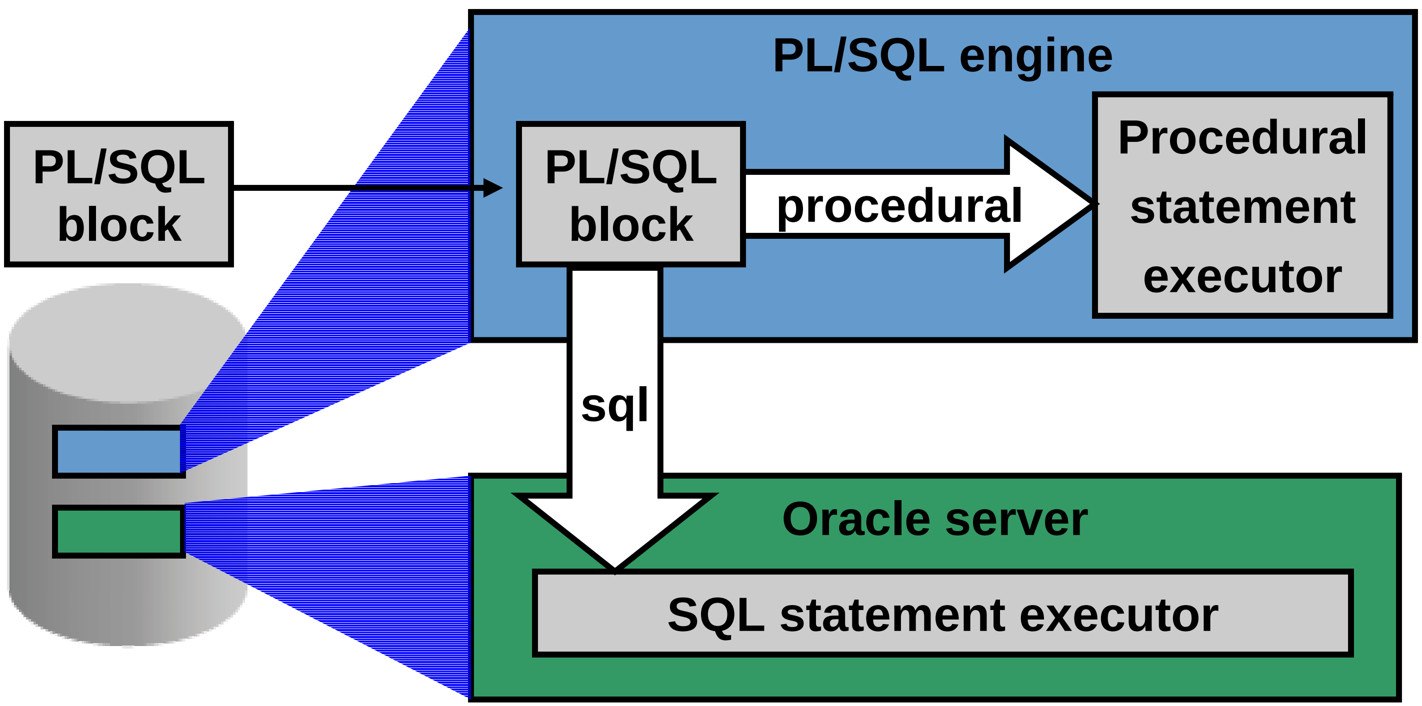
PL/SQL triggers are code blocks that execute when a specified application, database, or table event occurs.

- **Oracle Forms application triggers are standard anonymous blocks.**
- **Oracle database triggers have a specific structure.**

```
CREATE TRIGGER check_salary
BEFORE INSERT OR UPDATE ON employees
FOR EACH ROW
DECLARE
  c_min constant number(8,2) := 1000.0;
  c_max constant number(8,2) := 500000.0;
BEGIN
  IF :new.salary > c_max OR
     :new.salary < c_min THEN
    RAISE_APPLICATION_ERROR(-20000,
      'New salary is too small or large');
  END IF;
END;
/
```

PL/SQL Execution Environment

The PL/SQL run-time architecture:

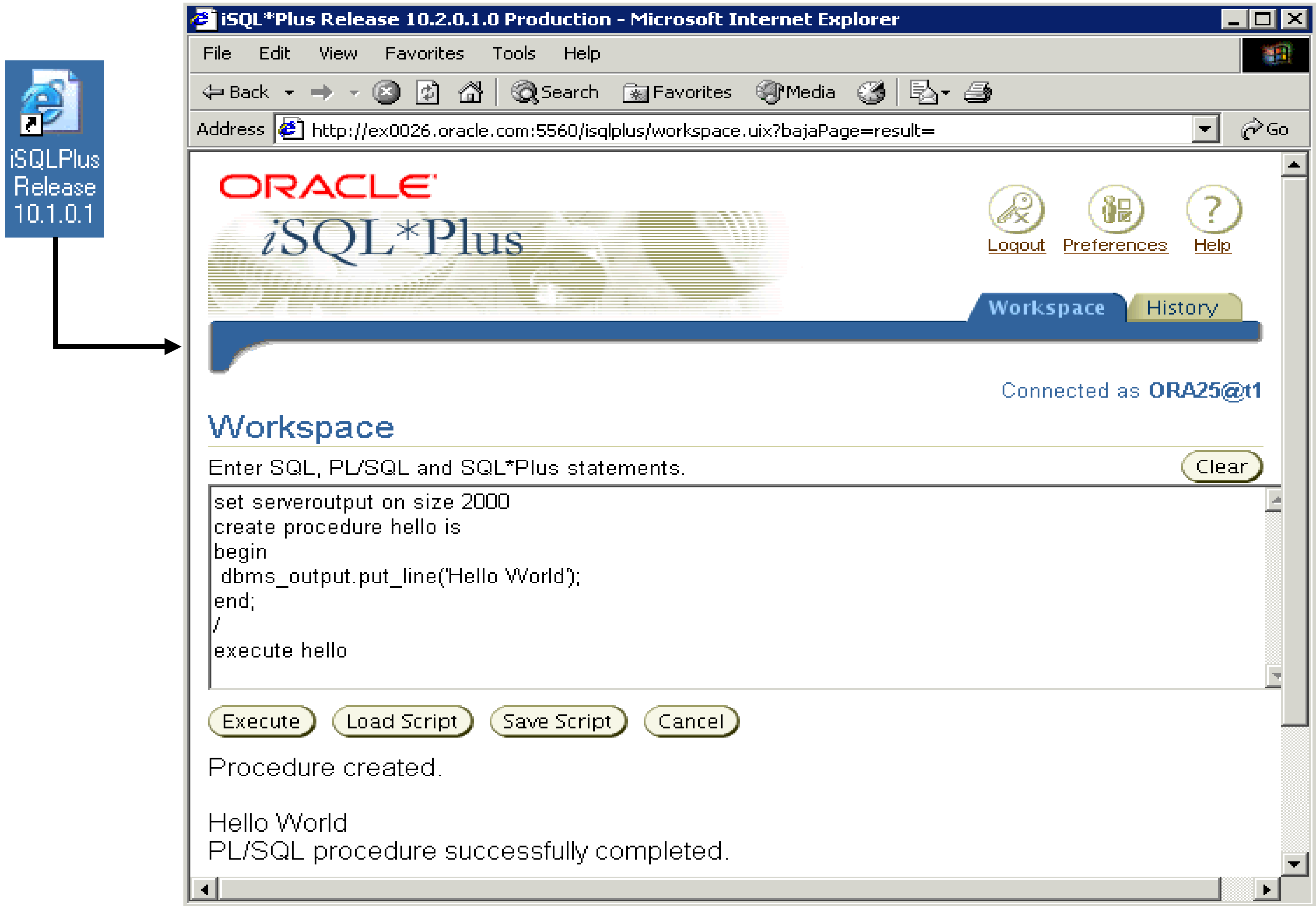


PL/SQL Development Environments

This course provides the following tools for developing PL/SQL code:

- **Oracle SQL*Plus (GUI or command-line versions)**
- **Oracle *i*SQL*Plus (used from a browser)**
- **Oracle JDeveloper IDE**

Coding PL/SQL in *i*SQL*Plus



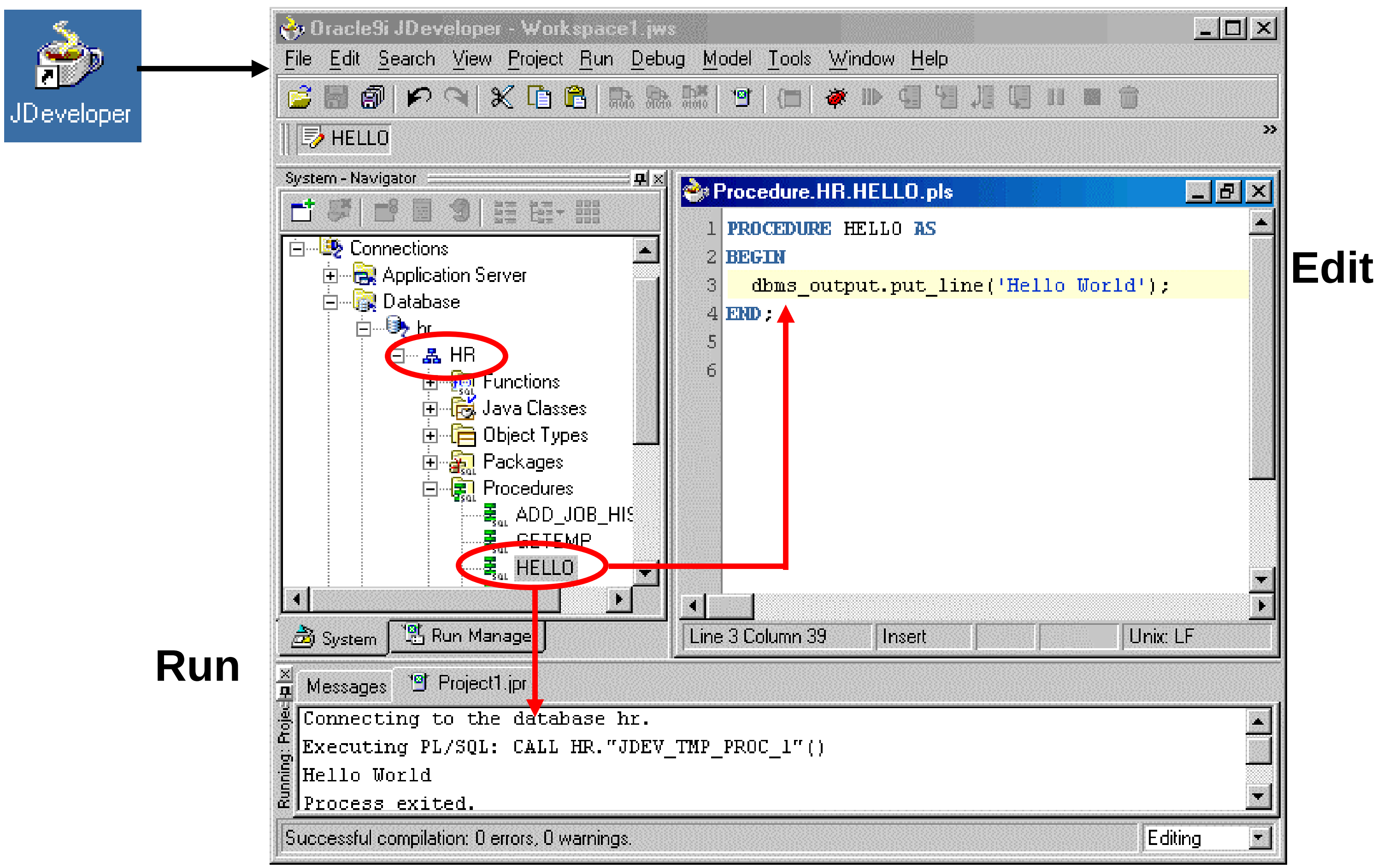
Coding PL/SQL in SQL*Plus

Oracle SQL*Plus

File Edit Search Options Help

```
SQL*Plus: Release 10.2.0.1.0 - Production on Tue Apr 4 12:10:08 2006

Copyright (c) 1982, 2005, Oracle.  All rights reserved.


Connected to:
Oracle Database 10g Enterprise Edition Release 10.2.0.1.0 - Production
With the Partitioning, Oracle Label Security, OLAP and Data Mining Scoring Engine options

SQL> set serveroutput on
SQL> create procedure hello is
  2  begin
  3   dbms_output.put_line('Hello World');
  4  end;
  5  /

Procedure created.

SQL> execute hello
Hello World

PL/SQL procedure successfully completed.

SQL>
```

Coding PL/SQL in Oracle JDeveloper



Summary

In this lesson, you should have learned how to:

- Declare named PL/SQL blocks, including procedures, functions, packages, and triggers
- Use anonymous (unnamed) PL/SQL blocks to invoke stored procedures and functions
- Use *i*SQL*Plus or SQL*Plus to develop PL/SQL code
- Explain the PL/SQL execution environment:
  - The client-side PL/SQL engine for executing PL/SQL code in Oracle Forms and Oracle Reports
  - The server-side PL/SQL engine for executing PL/SQL code stored in an Oracle database

Practice I: Overview

This practice covers the following topics:

- Browsing the HR tables
- Creating a simple PL/SQL procedure
- Creating a simple PL/SQL function
- Using an anonymous block to execute the PL/SQL procedure and function

1

Creating Stored Procedures

Objectives

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

- Describe and create a procedure
- Create procedures with parameters
- Differentiate between formal and actual parameters
- Use different parameter-passing modes
- Invoke a procedure
- Handle exceptions in procedures
- Remove a procedure

What Is a Procedure?

A procedure:

- **Is a type of subprogram that performs an action**
- **Can be stored in the database as a schema object**
- **Promotes reusability and maintainability**

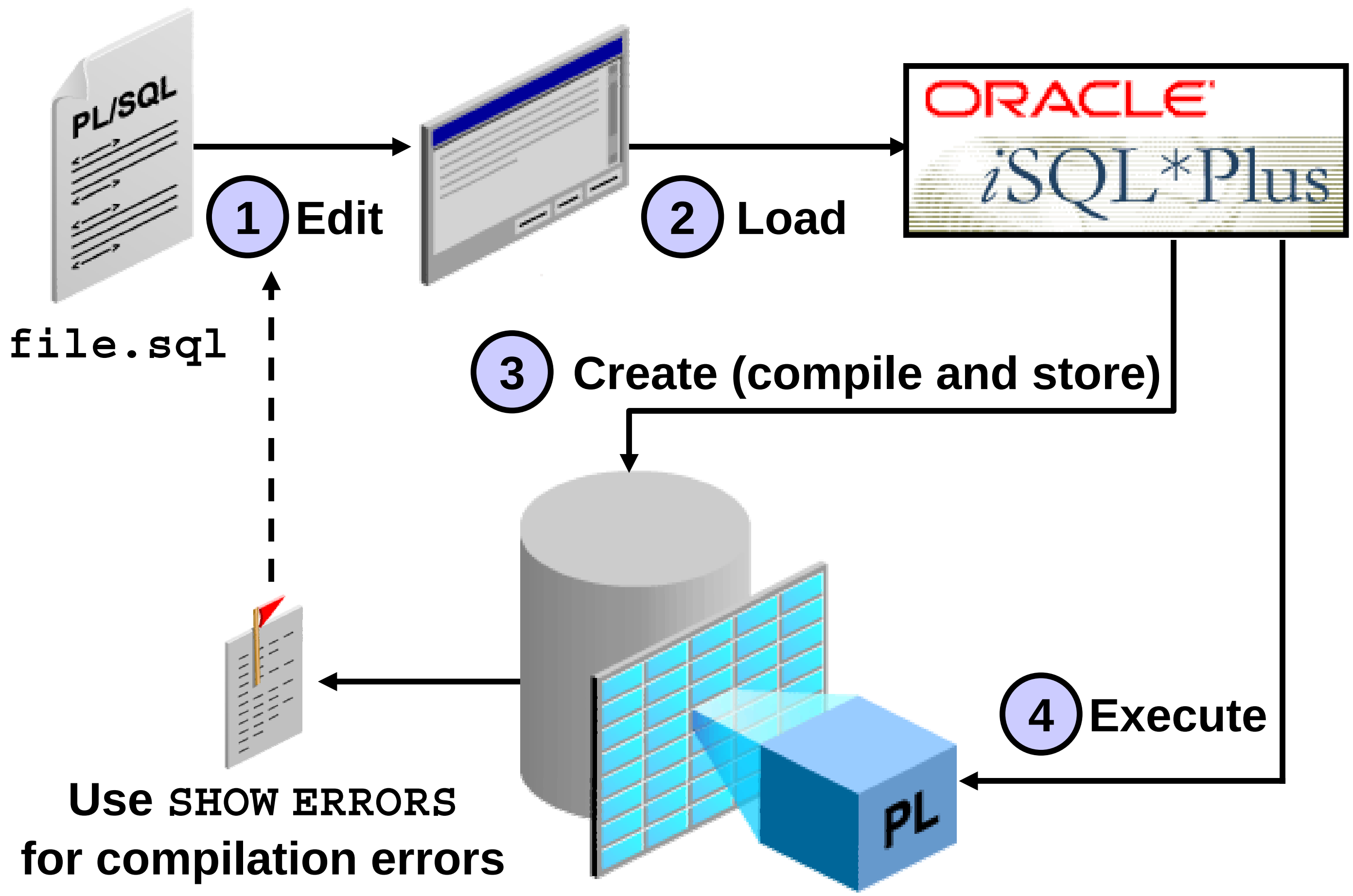
Syntax for Creating Procedures

- Use `CREATE PROCEDURE` followed by the name, optional parameters, and keyword `IS` or `AS`.
- Add the `OR REPLACE` option to overwrite an existing procedure.
- Write a PL/SQL block containing local variables, a `BEGIN` statement, and an `END` statement (or `END` *`procedure_name`*).

```
CREATE [OR REPLACE] PROCEDURE procedure_name
 [(parameter1 [mode] datatype1,
   parameter2 [mode] datatype2, ...)]
IS|AS
  [local_variable_declarations; …]
BEGIN
  -- actions;
END [procedure_name];
```

PL/SQL Block

Developing Procedures



What Are Parameters?

Parameters:

- **Are declared after the subprogram name in the PL/SQL header**
- **Pass or communicate data between the caller and the subprogram**
- **Are used like local variables but are dependent on their parameter-passing mode:**
  - **An `IN` parameter (the default) provides values for a subprogram to process.**
  - **An `OUT` parameter returns a value to the caller.**
  - **An `IN OUT` parameter supplies an input value, which may be returned (output) as a modified value.**

Formal and Actual Parameters

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

  Example:

```
CREATE PROCEDURE raise_sal(id NUMBER,sal NUMBER) IS
BEGIN ...
END raise_sal;
```

- **Actual parameters: Literal values, variables, and expressions used in the parameter list of the called subprogram**

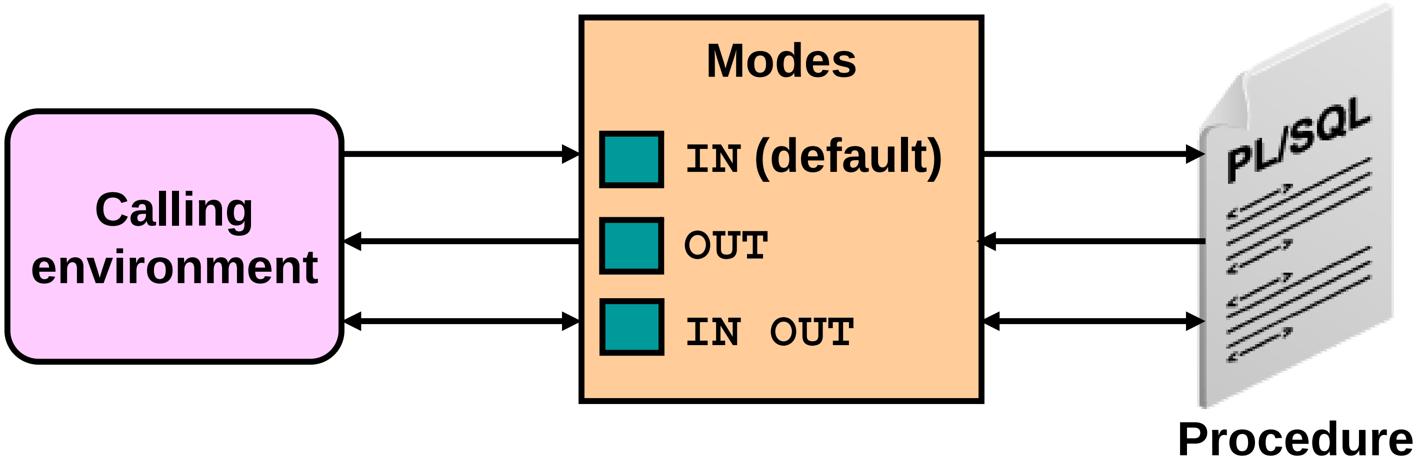
  Example:

```
emp_id := 100;
raise_sal(emp_id, 2000)
```

Procedural Parameter Modes

- Parameter modes are specified in the formal parameter declaration, after the parameter name and before its data type.
- The `IN` mode is the default if no mode is specified.

```
CREATE PROCEDURE procedure(param [mode] datatype)
...
```



Using IN Parameters: Example

```
CREATE OR REPLACE PROCEDURE raise_salary
  (id       IN employees.employee_id%TYPE,
   percent IN NUMBER)
IS
BEGIN
  UPDATE employees
  SET    salary = salary * (1 + percent/100)
  WHERE  employee_id = id;
END raise_salary;
/
```

```
EXECUTE raise_salary(176,10)
```

Using `OUT` Parameters: Example

```
CREATE OR REPLACE PROCEDURE query_emp
 (id      IN  employees.employee_id%TYPE,
  name    OUT employees.last_name%TYPE,
  salary  OUT employees.salary%TYPE) IS
BEGIN
  SELECT   last_name, salary INTO name, salary
   FROM    employees
   WHERE   employee_id = id;
END query_emp;
```

```
DECLARE
  emp_name employees.last_name%TYPE;
  emp_sal  employees.salary%TYPE;
BEGIN
  query_emp(171, emp_name, emp_sal); ...
END;
```

Viewing `OUT` Parameters with *i*SQL*Plus

- **Use PL/SQL variables that are printed with calls to the `DBMS_OUTPUT.PUT_LINE` procedure.**

```
SET SERVEROUTPUT ON
DECLARE
  emp_name employees.last_name%TYPE;
  emp_sal  employees.salary%TYPE;
BEGIN
  query_emp(171, emp_name, emp_sal);
  DBMS_OUTPUT.PUT_LINE('Name: ' || emp_name);
  DBMS_OUTPUT.PUT_LINE('Salary: ' || emp_sal);
END;
```

- **Use *i*SQL*Plus host variables, execute `QUERY_EMP` using host variables, and print the host variables.**

```
VARIABLE name VARCHAR2(25)
VARIABLE sal  NUMBER
EXECUTE query_emp(171, :name, :sal)
PRINT name sal
```

Calling PL/SQL Using Host Variables

A host variable (also known as a *bind* or a *global* variable):

- **Is declared and exists externally to the PL/SQL subprogram. A host variable can be created in:**
  - ***i*SQL*Plus by using the `VARIABLE` command**
  - **Oracle Forms internal and UI variables**
  - **Java variables**
- **Is preceded by a colon (`:`) when referenced in PL/SQL code**
- **Can be referenced in an anonymous block but not in a stored subprogram**
- **Provides a value to a PL/SQL block and receives a value from a PL/SQL block**

Using IN OUT Parameters: Example

Calling environment

phone_no (before the call)

'8006330575'

phone_no (after the call)

'(800)633-0575'

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

Syntax for Passing Parameters

- **Positional:**
  - Lists the actual parameters in the same order as the formal parameters
- **Named:**
  - Lists the actual parameters in arbitrary order and uses the association operator (=>) to associate a named formal parameter with its actual parameter
- **Combination:**
  - Lists some of the actual parameters as positional and some as named

Parameter Passing: Examples

```
CREATE OR REPLACE PROCEDURE add_dept(
  name IN departments.department_name%TYPE,
  loc  IN departments.location_id%TYPE) IS
BEGIN
  INSERT INTO departments(department_id,
              department_name, location_id)
  VALUES (departments_seq.NEXTVAL, name, loc);
END add_dept;
/
```

- **Passing by positional notation:**

```
EXECUTE add_dept ('TRAINING', 2500)
```

- **Passing by named notation:**

```
EXECUTE add_dept (loc=>2400, name=>'EDUCATION')
```

Using the DEFAULT Option for Parameters

- Defines default values for parameters:

```
CREATE OR REPLACE PROCEDURE add_dept(
 name departments.department_name%TYPE:='Unknown',
 loc  departments.location_id%TYPE DEFAULT 1700)
IS
BEGIN
  INSERT INTO departments (...)
  VALUES (departments_seq.NEXTVAL, name, loc);
END add_dept;
```

- Provides flexibility by combining the positional and named parameter-passing syntax:

```
EXECUTE add_dept
EXECUTE add_dept ('ADVERTISING', loc => 1200)
EXECUTE add_dept (loc => 1200)
```

ORACLE

Summary of Parameter Modes

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

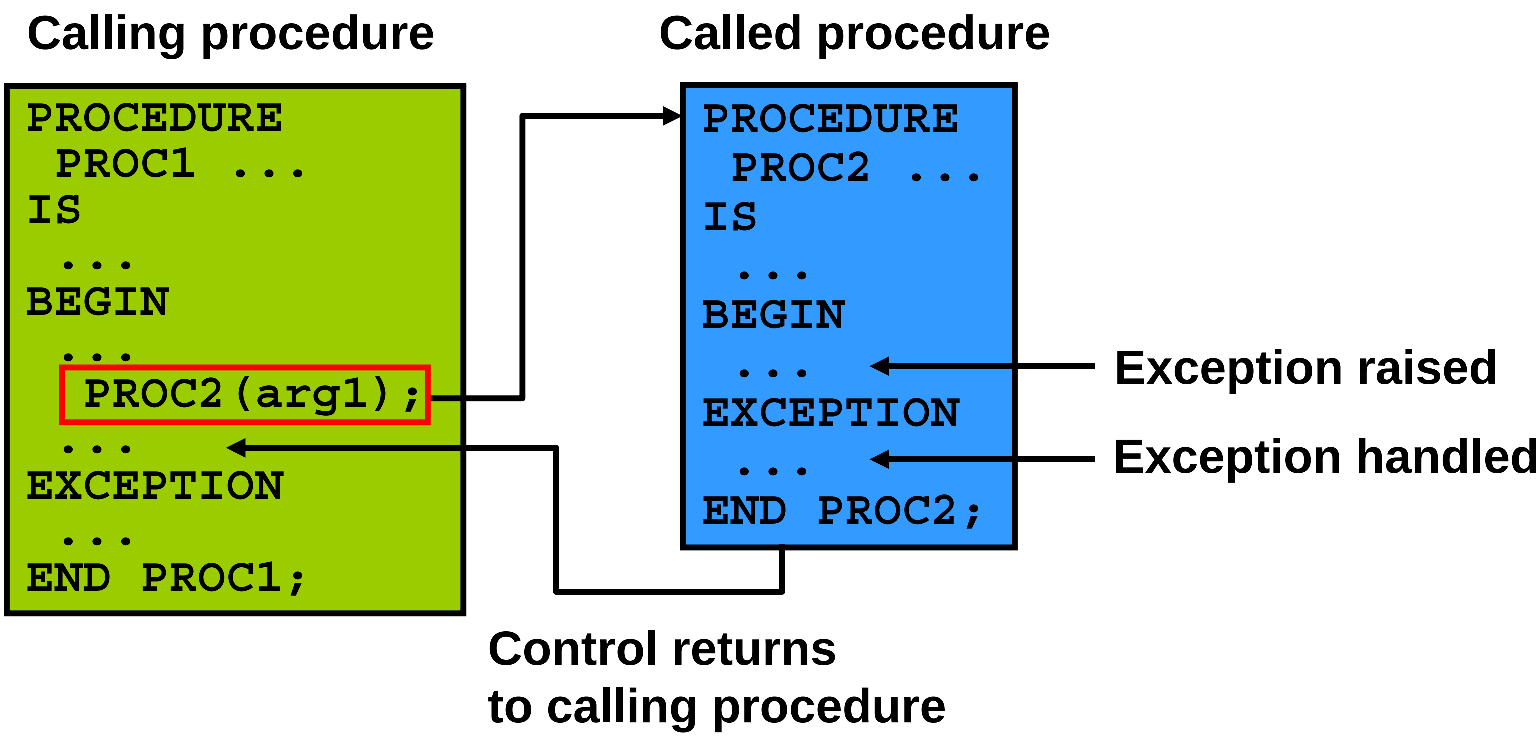
Invoking Procedures

You can invoke procedures by:

- Using anonymous blocks
- Using another procedure, as in the following example:

```
CREATE OR REPLACE PROCEDURE process_employees
IS
   CURSOR emp_cursor IS
    SELECT employee_id
    FROM   employees;
BEGIN
   FOR emp_rec IN emp_cursor
   LOOP
     raise_salary(emp_rec.employee_id, 10);
   END LOOP;
   COMMIT;
END process_employees;
/
```

Handled Exceptions

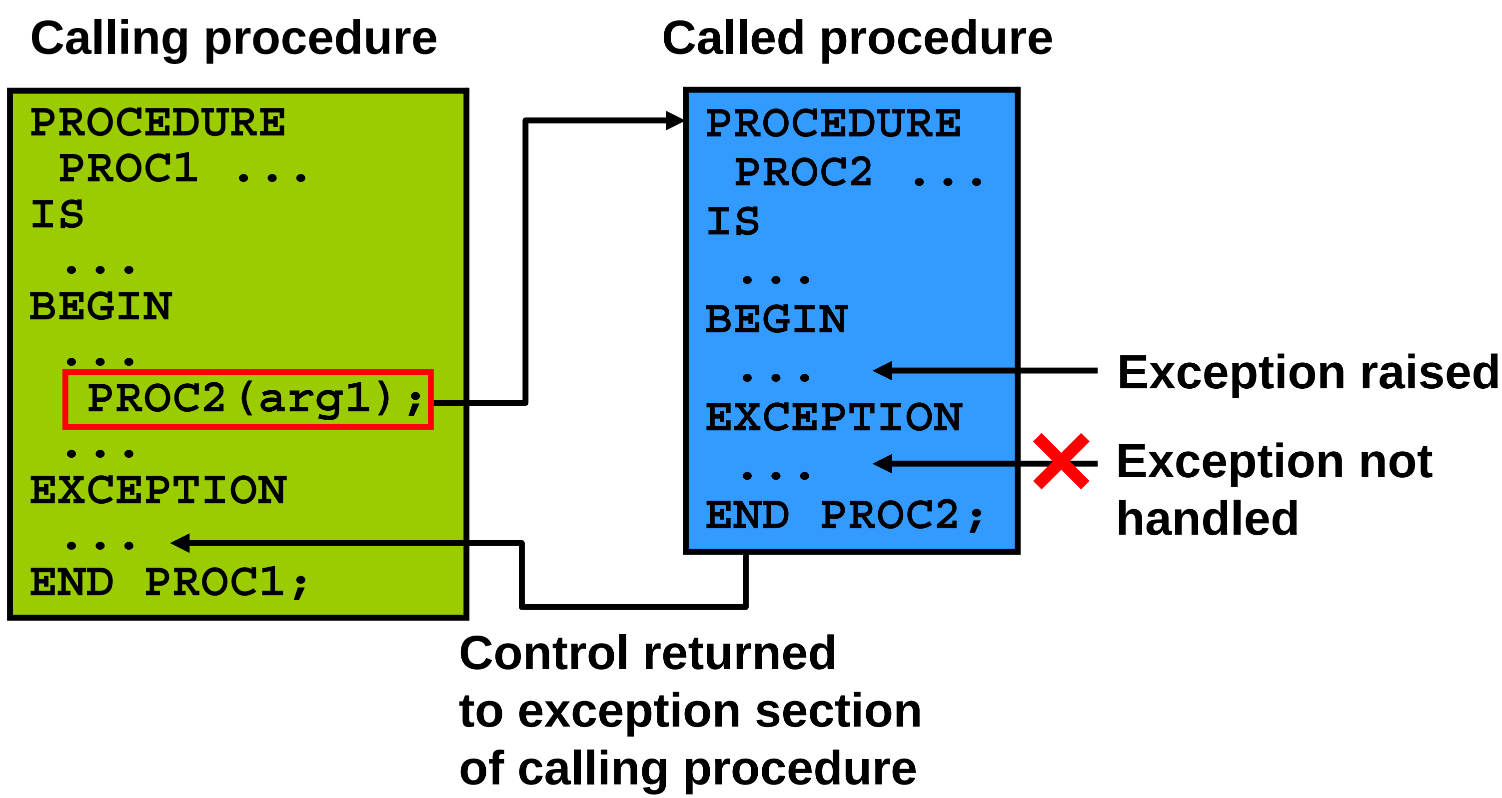


Handled Exceptions: Example

```
CREATE PROCEDURE add_department(
    name VARCHAR2, mgr NUMBER, loc NUMBER) IS
BEGIN
  INSERT INTO DEPARTMENTS (department_id,
    department_name, manager_id, location_id)
  VALUES (DEPARTMENTS_SEQ.NEXTVAL, name, mgr, loc);
  DBMS_OUTPUT.PUT_LINE('Added Dept: '||name);
EXCEPTION
 WHEN OTHERS THEN
  DBMS_OUTPUT.PUT_LINE('Err: adding dept: '||name);
END;
```

```
CREATE PROCEDURE create_departments IS
BEGIN
  add_department('Media', 100, 1800);
  add_department('Editing', 99, 1800);
  add_department('Advertising', 101, 1800);
END;
```

Exceptions Not Handled



Exceptions Not Handled: Example

```
SET SERVEROUTPUT ON
CREATE PROCEDURE add_department_noex(
    name VARCHAR2, mgr NUMBER, loc NUMBER) IS
BEGIN
  INSERT INTO DEPARTMENTS (department_id,
    department_name, manager_id, location_id)
  VALUES (DEPARTMENTS_SEQ.NEXTVAL, name, mgr, loc);
  DBMS_OUTPUT.PUT_LINE('Added Dept: '||name);
END;
```

```
CREATE PROCEDURE create_departments_noex IS
BEGIN
  add_department_noex('Media', 100, 1800);
  add_department_noex('Editing', 99, 1800);
  add_department_noex('Advertising', 101, 1800);
END;
```

Removing Procedures

You can remove a procedure that is stored in the database.

- **Syntax:**

```
DROP PROCEDURE procedure_name
```

- **Example:**

```
DROP PROCEDURE raise_salary;
```

Viewing Procedures in the Data Dictionary

Information for PL/SQL procedures is saved in the following data dictionary views:

- View source code in the `USER_SOURCE` table to view the subprograms that you own, or the `ALL_SOURCE` table for procedures that are owned by others who have granted you the `EXECUTE` privilege.

```
SELECT text
FROM   user_source
WHERE  name='ADD_DEPARTMENT' and type='PROCEDURE'
ORDER BY line;
```

- View the names of procedures in `USER_OBJECTS`.

```
SELECT object_name
FROM   user_objects
WHERE  object_type = 'PROCEDURE';
```

Benefits of Subprograms

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

Summary

In this lesson, you should have learned how to:

- Write a procedure to perform a task or an action
- Create, compile, and save procedures in the database by using the `CREATE PROCEDURE` SQL command
- Use parameters to pass data from the calling environment to the procedure by using three different parameter modes: `IN` (the default), `OUT`, and `IN OUT`
- Recognize the effect of handling and not handling exceptions on transactions and calling procedures

Summary

In this lesson, you should have learned how to:

- Remove procedures from the database by using the `DROP PROCEDURE` SQL command
- Modularize your application code by using procedures as building blocks

Practice 1: Overview

This practice covers the following topics:

- Creating stored procedures to:
  – Insert new rows into a table using the supplied parameter values
  – Update data in a table for rows that match the supplied parameter values
  – Delete rows from a table that match the supplied parameter values
  – Query a table and retrieve data based on supplied parameter values
- Handling exceptions in procedures
- Compiling and invoking procedures

2

Creating Stored Functions

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

ORACLE

Overview of Stored Functions

A function:

- Is a named PL/SQL block that returns a value
- Can be stored in the database as a schema object for repeated execution
- Is called as part of an expression or is used to provide a parameter value

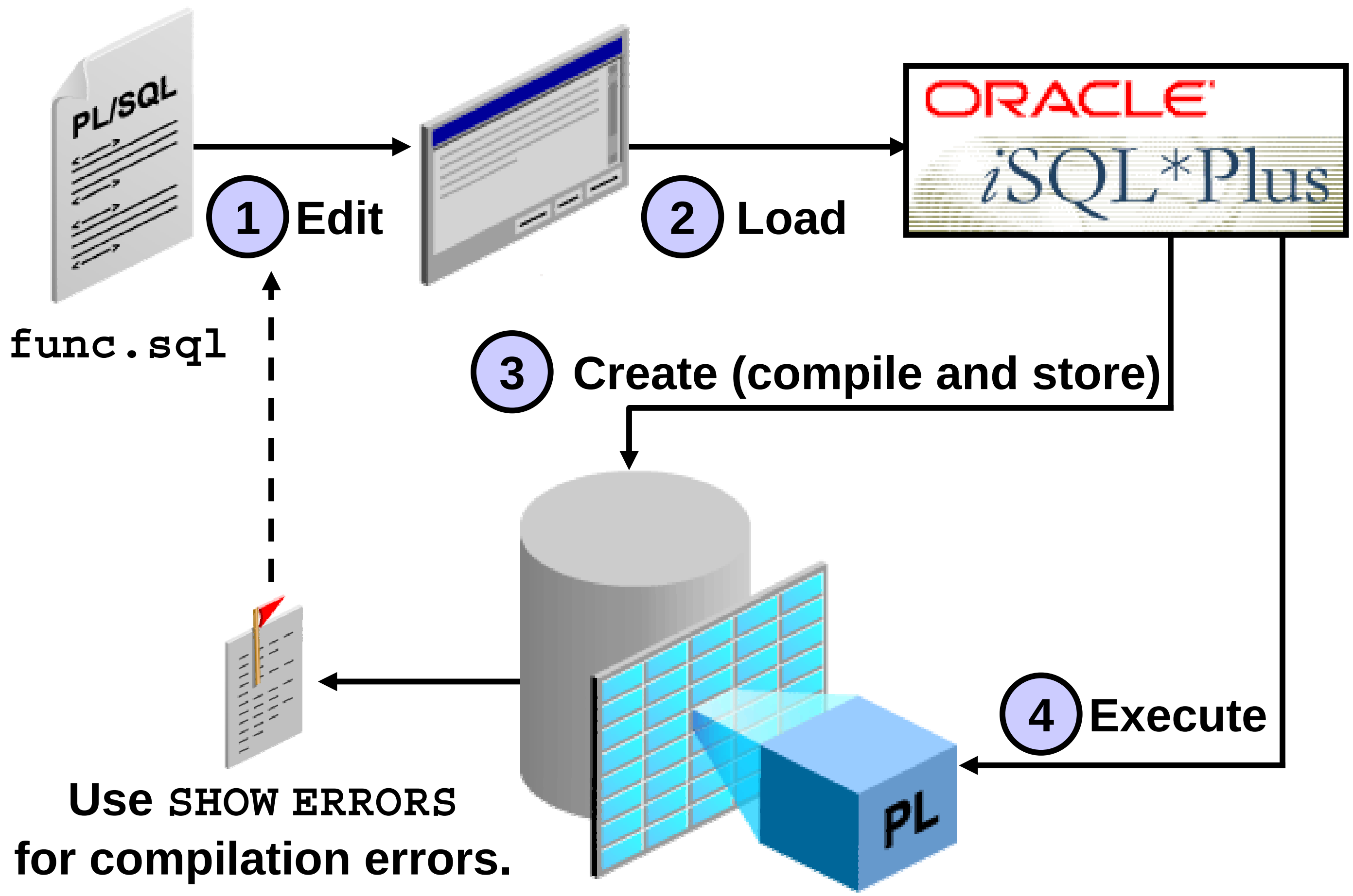
Syntax for Creating Functions

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

```
CREATE [OR REPLACE] FUNCTION function_name
 [(parameter1 [mode1] datatype1, ...)]
RETURN datatype IS|AS
 [local_variable_declarations; …]
BEGIN
  -- actions;
  RETURN expression;
END [function_name];
```

PL/SQL Block

Developing Functions



Stored Function: Example

- Create the function:

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

- Invoke the function as an expression or as a parameter value:

```
EXECUTE dbms_output.put_line(get_sal(100))
```

Ways to Execute Functions

- Invoke as part of a PL/SQL expression
  - Using a host variable to obtain the result:

```
VARIABLE salary NUMBER
EXECUTE :salary := get_sal(100)
```

  - Using a local variable to obtain the result:

```
DECLARE sal employees.salary%type;
BEGIN
  sal := get_sal(100); ...
END;
```

- Use as a parameter to another subprogram

```
EXECUTE dbms_output.put_line(get_sal(100))
```

- Use in a SQL statement (subject to restrictions)

```
SELECT job_id, get_sal(employee_id) FROM employees;
```

Advantages of User-Defined Functions in SQL Statements

- Can 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 data values

ORACLE

Function in SQL Expressions: Example

```
CREATE OR REPLACE FUNCTION tax(value IN NUMBER)
 RETURN NUMBER IS
BEGIN
   RETURN (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

User-defined functions act like built-in single-row functions and can be used in:

- The `SELECT` list or clause of a query
- Conditional expressions of the `WHERE` and `HAVING` clauses
- The `CONNECT BY`, `START WITH`, `ORDER BY`, and `GROUP BY` clauses of a query
- The `VALUES` clause of the `INSERT` statement
- The `SET` clause of the `UPDATE` statement

ORACLE

Restrictions on Calling Functions from SQL Expressions

- User-defined functions that are callable from SQL expressions must:
  – Be stored in the database
  – Accept only `IN` parameters with valid SQL data types, not PL/SQL-specific types
  – Return valid SQL data types, not PL/SQL-specific types
- When calling functions in SQL statements:
  – Parameters must be specified with positional notation
  – You must own the function or have the `EXECUTE` privilege

Controlling Side Effects When Calling Functions from SQL Expressions

Functions called from:

- **A `SELECT` statement cannot contain DML statements**
- **An `UPDATE` or `DELETE` statement on a table `T` cannot query or contain DML on the same table `T`**
- **SQL statements cannot end transactions (that is, cannot execute `COMMIT` or `ROLLBACK` operations)**

Note: Calls to subprograms that break these restrictions are also not allowed in the function.

Restrictions on Calling Functions from SQL: Example

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

```
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

Removing a stored function:

- **You can drop a stored function by using the following syntax:**

```
DROP FUNCTION function_name
```

Example:

```
DROP FUNCTION get_sal;
```

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

ORACLE

Viewing Functions in the Data Dictionary

Information for PL/SQL functions is stored in the following Oracle data dictionary views:

- You can view source code in the `USER_SOURCE` table for subprograms that you own, or the `ALL_SOURCE` table for functions owned by others who have granted you the `EXECUTE` privilege.

```
SELECT text
FROM   user_source
WHERE  type = 'FUNCTION'
ORDER BY line;
```

- You can view the names of functions by using `USER_OBJECTS`.

```
SELECT object_name
FROM   user_objects
WHERE  object_type = 'FUNCTION';
```

Procedures Versus 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 values (if any) in output parameters	Must return a single value
Can contain a `RETURN` statement without a value	Must contain at least one `RETURN` statement

Summary

In this lesson, you should have learned how to:

- Write a PL/SQL function to compute and return a value by using the `CREATE FUNCTION` SQL statement
- Invoke a function as part of a PL/SQL expression
- Use stored PL/SQL functions in SQL statements
- Remove a function from the database by using the `DROP FUNCTION` SQL statement

Practice 2: Overview

This practice covers the following topics:

- **Creating stored functions:**
  - **To query a database table and return specific values**
  - **To be used in a SQL statement**
  - **To insert a new row, with specified parameter values, into a database table**
  - **Using default parameter values**
- **Invoking a stored function from a SQL statement**
- **Invoking a stored function from a stored procedure**

ORACLE

3

Creating Packages

Objectives

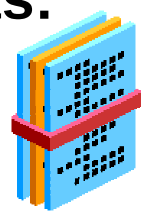
After completing this lesson, you should be able to do the following:

- Describe packages and list their 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 the use of a bodiless package

PL/SQL Packages: Overview

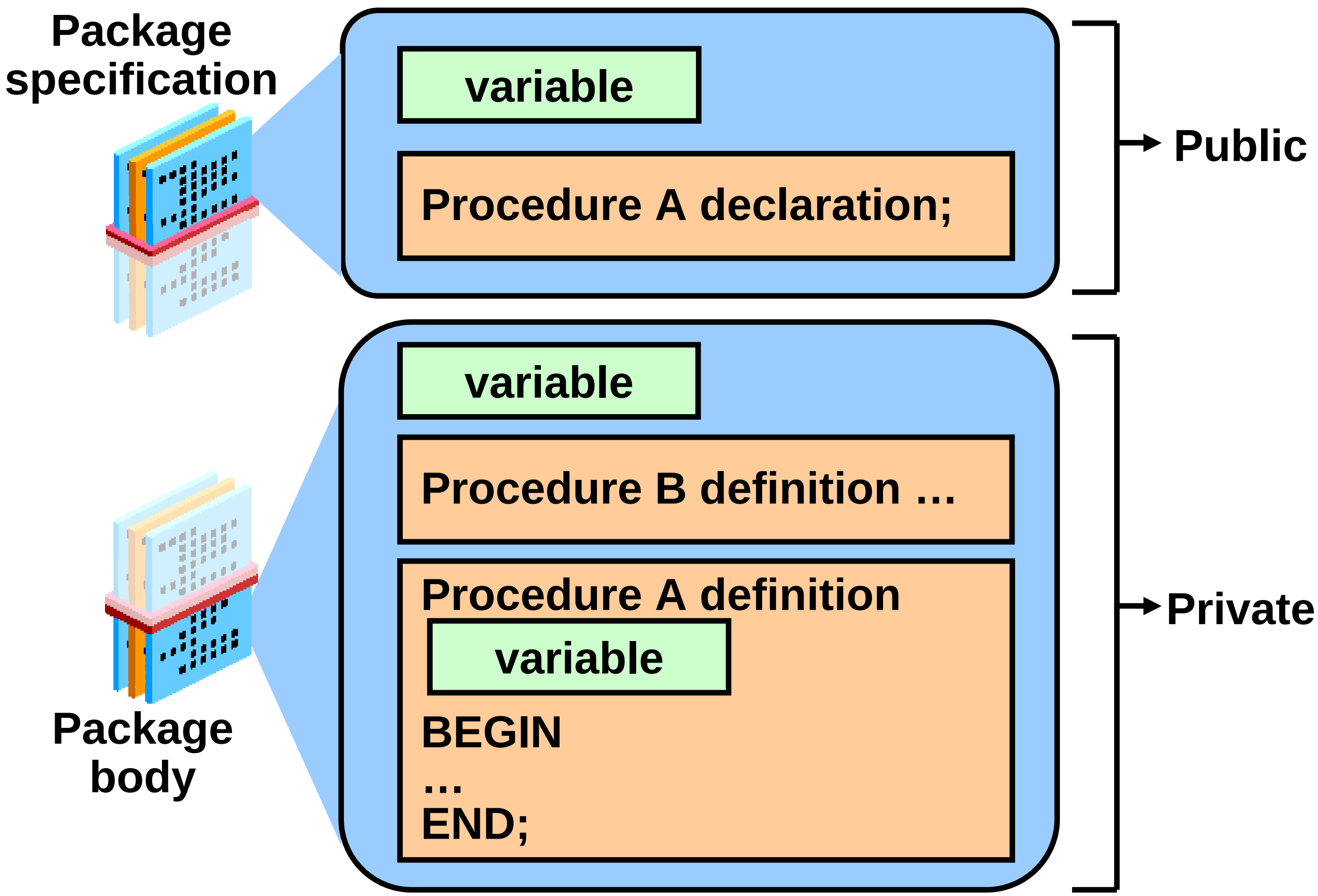
PL/SQL packages:

- **Group logically related components:**
  - **PL/SQL types**
  - **Variables, data structures, and exceptions**
  - **Subprograms: Procedures and functions**
- **Consist of two parts:**
  - **A specification**
  - **A body**

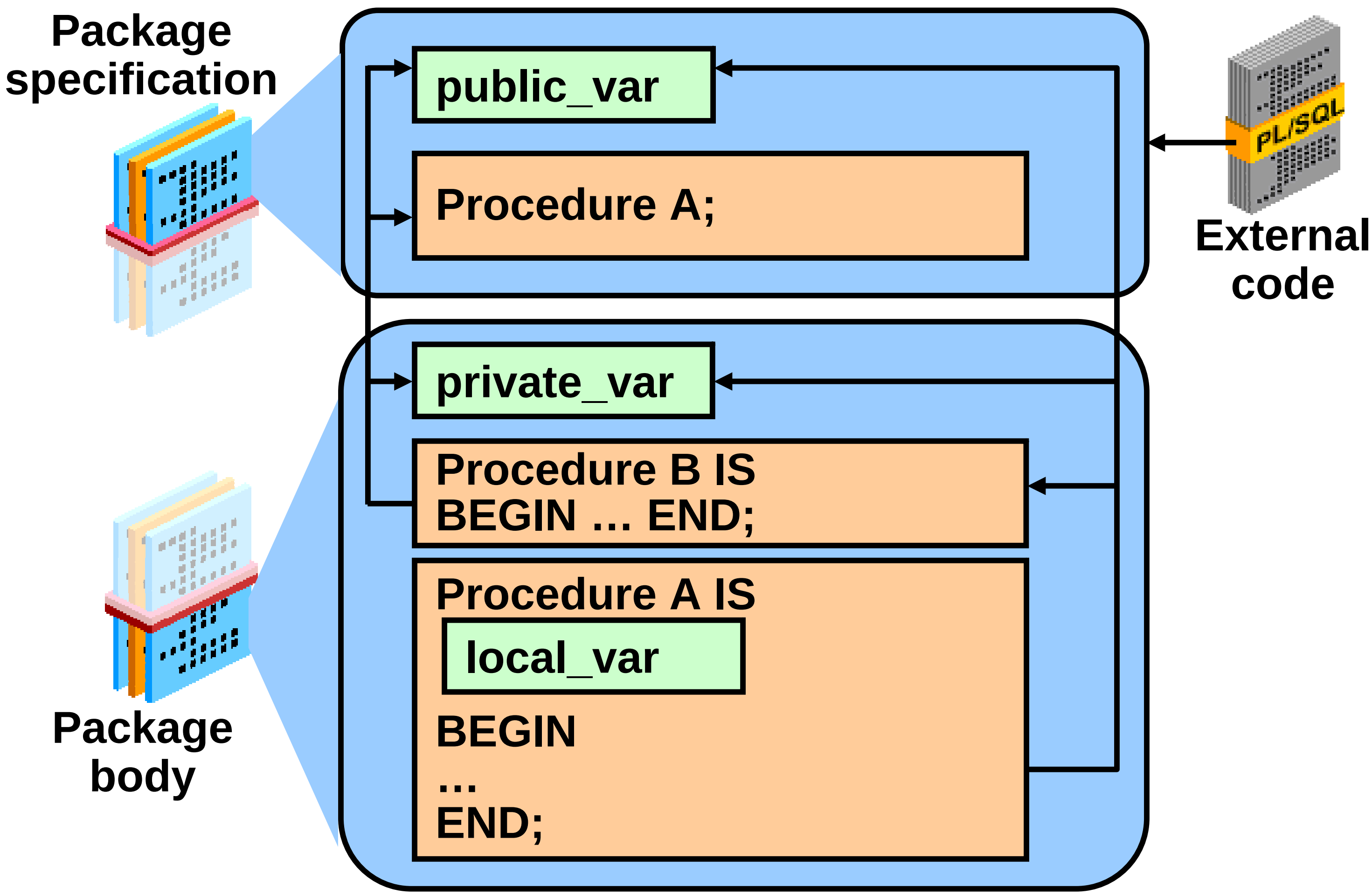


- **Enable the Oracle server to read multiple objects into memory at once**

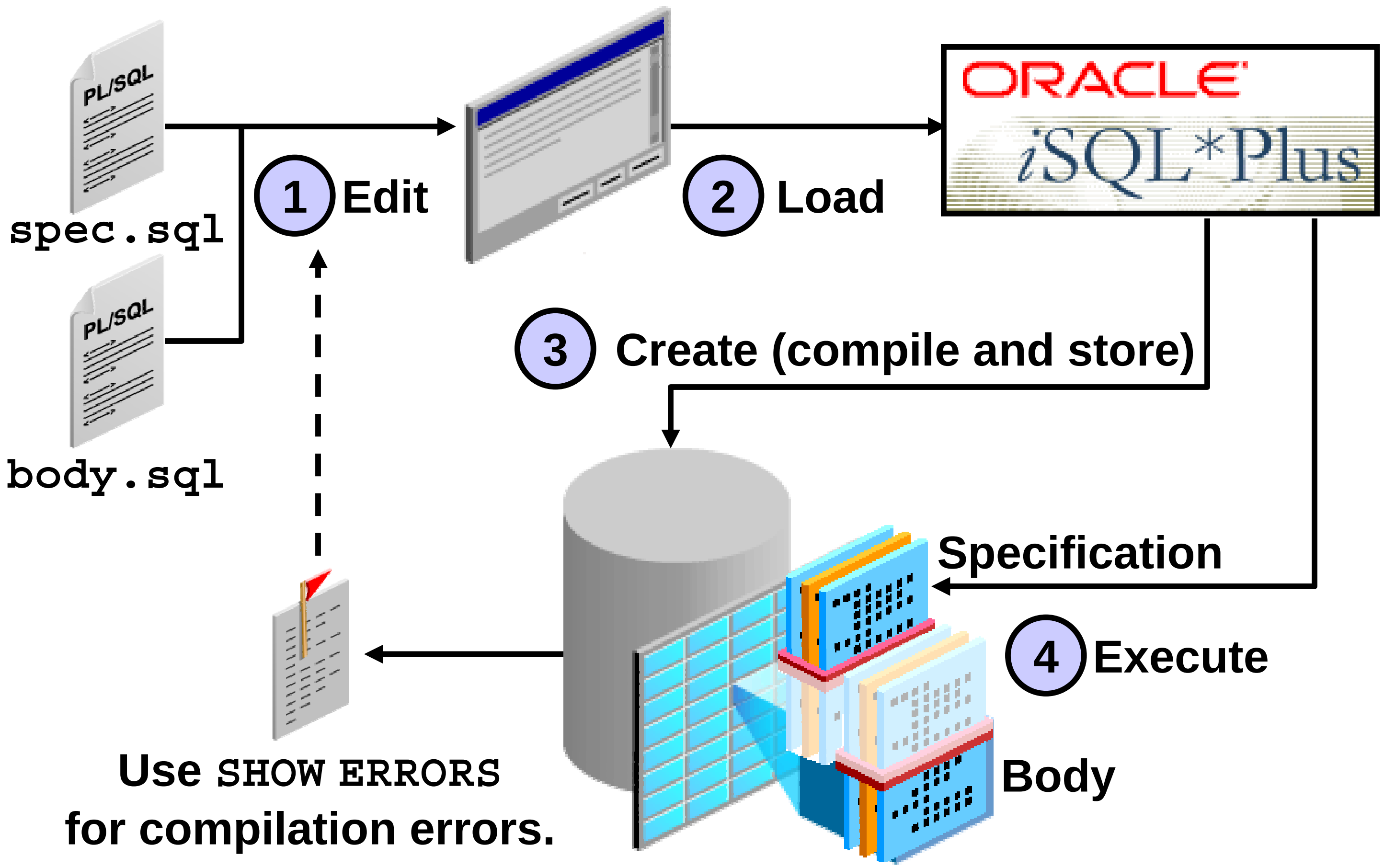
Components of a PL/SQL Package



Visibility of Package Components



Developing PL/SQL Packages



Creating the Package Specification

Syntax:

```
CREATE [OR REPLACE] PACKAGE package_name IS|AS
    public type and variable declarations
    subprogram specifications
END [package_name];
```

- **The `OR REPLACE` option drops and re-creates 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.**

Example of Package Specification: `comm_pkg`

```
CREATE OR REPLACE PACKAGE comm_pkg IS
  std_comm NUMBER := 0.10;  --initialized to 0.10
  PROCEDURE reset_comm(new_comm NUMBER);
END comm_pkg;
/
```

- **`STD_COMM` is a global variable initialized to `0.10`.**
- **`RESET_COMM` is a public procedure used to reset the standard commission based on some business rules. It is implemented in the package body.**

Creating the Package Body

Syntax:

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

- **The `OR REPLACE` option drops and re-creates the package body.**
- **Identifiers defined in the package body are private and not visible outside the package body.**
- **All private constructs must be declared before they are referenced.**
- **Public constructs are visible to the package body.**

Example of Package Body: comm_pkg

```
CREATE OR REPLACE PACKAGE BODY comm_pkg IS
  FUNCTION validate(comm NUMBER) RETURN BOOLEAN IS
    max_comm employees.commission_pct%type;
  BEGIN
    SELECT MAX(commission_pct) INTO max_comm
    FROM   employees;
    RETURN (comm BETWEEN 0.0 AND max_comm);
  END validate;
  PROCEDURE reset_comm (new_comm NUMBER) IS BEGIN
    IF validate(new_comm) THEN
      std_comm := new_comm; -- reset public var
    ELSE  RAISE_APPLICATION_ERROR(
            -20210, 'Bad Commission');
    END IF;
  END reset_comm;
END comm_pkg;
```

Invoking Package Subprograms

- **Invoke a function within the same package:**

```
CREATE OR REPLACE PACKAGE BODY comm_pkg IS ...
  PROCEDURE reset_comm(new_comm NUMBER) IS
  BEGIN
    IF validate(new_comm) THEN
      std_comm := new_comm;
    ELSE ...
    END IF;
  END reset_comm;
END comm_pkg;
```

- **Invoke a package procedure from *i*SQL*Plus:**

```
EXECUTE comm_pkg.reset_comm(0.15)
```

- **Invoke a package procedure in a different schema:**

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

Creating and Using Bodiless Packages

```
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;
```

```
BEGIN  DBMS_OUTPUT.PUT_LINE('20 miles = ' ||
        20 * global_consts.mile_2_kilo || ' km');
END;
```

```
CREATE FUNCTION mtr2yrd(m NUMBER) RETURN NUMBER IS
BEGIN
  RETURN (m * global_consts.meter_2_yard);
END mtr2yrd;
/
EXECUTE DBMS_OUTPUT.PUT_LINE(mtr2yrd(1))
```

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;
```

Viewing Packages in the Data Dictionary

The source code for PL/SQL packages is maintained and is viewable through the `USER_SOURCE` and `ALL_SOURCE` tables in the data dictionary.

- To view the package specification, use:

```
SELECT text
FROM   user_source
WHERE  name = 'COMM_PKG' AND type = 'PACKAGE';
```

- To view the package body, use:

```
SELECT text
FROM   user_source
WHERE  name = 'COMM_PKG' AND type = 'PACKAGE BODY';
```

Guidelines for Writing 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.

ORACLE

Advantages of Using Packages

- Modularity: Encapsulating related constructs
- Easier maintenance: Keeping logically related functionality together
- Easier application design: Coding and compiling the 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 Using 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 code organization, management, security, and performance by using packages
- Create and remove package specifications and bodies
- Group related procedures and functions together in a package
- Encapsulate the code in a package body
- Define and use components in bodiless packages
- Change a package body without affecting a package specification

ORACLE

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 package body.
`DROP PACKAGE BODY`	Remove only the package body.

ORACLE

Practice 3: Overview

This practice covers the following topics:

- **Creating packages**
- **Invoking package program units**

4

Using More Package Concepts

Objectives

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

- Overload package procedures and functions
- Use forward declarations
- Create an initialization block in a package body
- Manage persistent package data states for the life of a session
- Use PL/SQL tables and records in packages
- Wrap source code stored in the data dictionary so that it is not readable

Overloading Subprograms

The overloading feature in PL/SQL:

- Enables you to create two or more subprograms with the same name
- Requires that the subprogram's formal parameters differ in number, order, or data type family
- Enables you to build flexible ways for invoking subprograms with different data
- Provides a way to extend functionality without loss of existing code

Note: Overloading can be done with local subprograms, package subprograms, and type methods, but not with stand-alone subprograms.

Overloading: Example

```
CREATE OR REPLACE PACKAGE dept_pkg IS
  PROCEDURE add_department(deptno NUMBER,
   name VARCHAR2 := 'unknown', loc NUMBER := 1700);
  PROCEDURE add_department(
   name VARCHAR2 := 'unknown', loc NUMBER := 1700);
END dept_pkg;
/
```

Overloading: Example

```
CREATE OR REPLACE PACKAGE BODY dept_pkg  IS
  PROCEDURE add_department (deptno NUMBER,
   name VARCHAR2:='unknown', loc NUMBER:=1700) IS
  BEGIN
    INSERT INTO departments(department_id,
      department_name, location_id)
    VALUES  (deptno, name, loc);
  END add_department;

  PROCEDURE add_department (
    name VARCHAR2:='unknown', loc NUMBER:=1700) IS
  BEGIN
    INSERT INTO departments (department_id,
      department_name, location_id)
    VALUES (departments_seq.NEXTVAL, name, loc);
  END add_department;
 END dept_pkg;
/
```

Overloading and the STANDARD Package

- **A package named `STANDARD` defines the PL/SQL environment and built-in functions.**
- **Most built-in functions are overloaded. An example is the `TO_CHAR` function:**

```
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;
```

- **A PL/SQL subprogram with the same name as a built-in subprogram overrides the standard declaration in the local context, unless you qualify the built-in subprogram with its package name.**

Using Forward Declarations

- Block-structured languages (such as PL/SQL) must declare identifiers before referencing them.
- Example of a referencing problem:

```
CREATE OR REPLACE PACKAGE BODY forward_pkg IS
  PROCEDURE award_bonus(. . .) IS
  BEGIN

    calc_rating (. . .);      --illegal reference

  END;

  PROCEDURE calc_rating (. . .) IS
  BEGIN
    ...
  END;
END forward_pkg;
/
```

Using Forward Declarations

In the package body, a forward declaration is a private subprogram specification terminated by a semicolon.

```
CREATE OR REPLACE PACKAGE BODY forward_pkg IS
 PROCEDURE calc_rating (...);-- forward declaration

 -- Subprograms defined in alphabetical order

 PROCEDURE award_bonus(...) IS
 BEGIN
   calc_rating (...);         -- reference resolved!
    . . .
 END;

 PROCEDURE calc_rating (...) IS -- implementation
 BEGIN
    . . .
 END;
END forward_pkg;
```

Package Initialization Block

The block at the end of the package body executes once and is used to initialize public and private package variables.

```
CREATE OR REPLACE PACKAGE taxes IS
  tax    NUMBER;
  ...   -- declare all public procedures/functions
END taxes;
/
CREATE OR REPLACE PACKAGE BODY taxes IS
  ... -- declare all private variables
  ... -- define public/private procedures/functions
 BEGIN
   SELECT   rate_value INTO tax
   FROM     tax_rates
   WHERE    rate_name = 'TAX';
END taxes;
/
```

Using Package Functions in SQL and Restrictions

- Package functions can be used in SQL statements.
- Functions called from:
  - A query or DML statement must not end the current transaction, create or roll back to a savepoint, or alter the system or session
  - A query or a parallelized DML statement cannot execute a DML statement or modify the database
  - A DML statement cannot read or modify the table being changed by that DML statement

Note: A function calling subprograms that break the preceding restrictions is not allowed.

ORACLE

Package Function in SQL: Example

```
CREATE OR REPLACE PACKAGE taxes_pkg IS
  FUNCTION tax (value IN NUMBER) RETURN NUMBER;
END taxes_pkg;
/
CREATE OR REPLACE PACKAGE BODY taxes_pkg IS
  FUNCTION tax (value IN NUMBER) RETURN NUMBER IS
    rate NUMBER := 0.08;
  BEGIN
    RETURN (value * rate);
  END tax;
END taxes_pkg;
/
```

```
SELECT taxes_pkg.tax(salary), salary, last_name
FROM   employees;
```

Persistent State of Packages

The collection of package variables and the values define the package state. The package state is:

- Initialized when the package is first loaded
- Persistent (by default) for the life of the session
  - Stored in the User Global Area (UGA)
  - Unique to each session
  - Subject to change when package subprograms are called or public variables are modified
- Not persistent for the session but persistent for the life of a subprogram call when using `PRAGMA SERIALLY_REUSABLE` in the package specification

Persistent State of Package Variables: Example

Time	Events	Scott STD	Scott MAX	Jones STD	Jones MAX
9:00	Scott> EXECUTE comm_pkg.reset_comm(0.25)	0.10 0.25	0.4	-	0.4
9:30	Jones> INSERT INTO employees(last_name,commission_pct) VALUES('Madonna', 0.8);	0.25	0.4		0.8
9:35	Jones> EXECUTE comm_pkg.reset_comm (0.5)	0.25	0.4	0.1 0.5	0.8
10:00	Scott> EXECUTE comm_pkg.reset_comm(0.6) Err -20210 'Bad Commission'	0.25	0.4	0.5	0.8
11:00	Jones> ROLLBACK;	0.25	0.4	0.5	0.4
11:01	EXIT ...	0.25	0.4	-	0.4
12:00	EXEC comm_pkg.reset_comm(0.2)	0.25	0.4	0.2	0.4

Persistent State of a Package Cursor

```
CREATE OR REPLACE PACKAGE BODY curs_pkg IS
  CURSOR c IS SELECT employee_id FROM employees;
  PROCEDURE open IS
  BEGIN
    IF NOT c%ISOPEN THEN   OPEN c;  END IF;
  END open;
  FUNCTION next(n NUMBER := 1) RETURN BOOLEAN IS
    emp_id employees.employee_id%TYPE;
  BEGIN
    FOR count IN 1 .. n LOOP
      FETCH c INTO emp_id;
      EXIT WHEN c%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE('Id: ' ||(emp_id));
    END LOOP;
    RETURN c%FOUND;
  END next;
  PROCEDURE close IS BEGIN
    IF c%ISOPEN THEN  CLOSE c;  END IF;
  END close;
END curs_pkg;
```

Executing `CURS_PKG`

```
SET SERVEROUTPUT ON
EXECUTE curs_pkg.open
DECLARE
  more BOOLEAN := curs_pkg.next(3);
BEGIN
  IF NOT more THEN
    curs_pkg.close;
  END IF;
END;
/
RUN  -- repeats execution on the anonymous block
EXECUTE curs_pkg.close
```

Using PL/SQL Tables of Records in Packages

```
CREATE OR REPLACE PACKAGE emp_pkg IS
  TYPE emp_table_type IS TABLE OF employees%ROWTYPE
     INDEX BY BINARY_INTEGER;
  PROCEDURE get_employees(emps OUT emp_table_type);
END emp_pkg;
/
```

```
CREATE OR REPLACE PACKAGE BODY emp_pkg IS
  PROCEDURE get_employees(emps OUT emp_table_type) IS
    i BINARY_INTEGER := 0;
  BEGIN
    FOR emp_record IN (SELECT * FROM employees)
    LOOP
      emps(i) := emp_record;
      i:= i+1;
    END LOOP;
  END get_employees;
END emp_pkg;
/
```

PL/SQL Wrapper

- The PL/SQL wrapper is a stand-alone utility that hides application internals by converting PL/SQL source code into portable object code.
- Wrapping has the following features:
  - Platform independence
  - Dynamic loading
  - Dynamic binding
  - Dependency checking
  - Normal importing and exporting when invoked

Running the Wrapper

The command-line syntax is:

```
WRAP INAME=input_file_name [ONAME=output_file_name]
```

- The `INAME` argument is required.
- The default extension for the input file is `.sql`, unless it is specified with the name.
- The `ONAME` argument is optional.
- The default extension for output file is `.plb`, unless specified with the `ONAME` argument.

Examples:

```
WRAP INAME=demo_04_hello.sql
WRAP INAME=demo_04_hello
WRAP INAME=demo_04_hello.sql ONAME=demo_04_hello.plb
```

Results of Wrapping

- **Original PL/SQL source code in input file:**

```
CREATE PACKAGE banking IS
  min_bal := 100;
  no_funds EXCEPTION;
...
END banking;
/
```

- **Wrapped code in output file:**

```
CREATE PACKAGE banking
wrapped

012abc463e ...

/
```

Guidelines for Wrapping

- **You must wrap only the package body, not the package specification.**
- **The wrapper can detect syntactic errors but cannot detect semantic errors.**
- **The output file should not be edited. You maintain the original source code and wrap again as required.**

Summary

In this lesson, you should have learned how to:

- Create and call overloaded subprograms
- Use forward declarations for subprograms
- Write package initialization blocks
- Maintain persistent package state
- Use the PL/SQL wrapper to wrap code

Practice 4: Overview

This practice covers the following topics:

- Using overloaded subprograms
- Creating a package initialization block
- Using a forward declaration
- Using the `WRAP` utility to prevent the source code from being deciphered by humans

5

Using Oracle-Supplied Packages in Application Development

Objectives

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

- Describe how the `DBMS_OUTPUT` package works
- Use `UTL_FILE` to direct output to operating system files
- Use the `HTP` package to generate a simple Web page
- Describe the main features of `UTL_MAIL`
- Call the `DBMS_SCHEDULER` package to schedule PL/SQL code for execution

Using Oracle-Supplied Packages

The Oracle-supplied packages:

- Are provided with the Oracle server
- Extend the functionality of the database
- Enable access to certain SQL features that are normally restricted for PL/SQL

For example, the `DBMS_OUTPUT` package was originally designed to debug PL/SQL programs.

List of Some Oracle-Supplied Packages

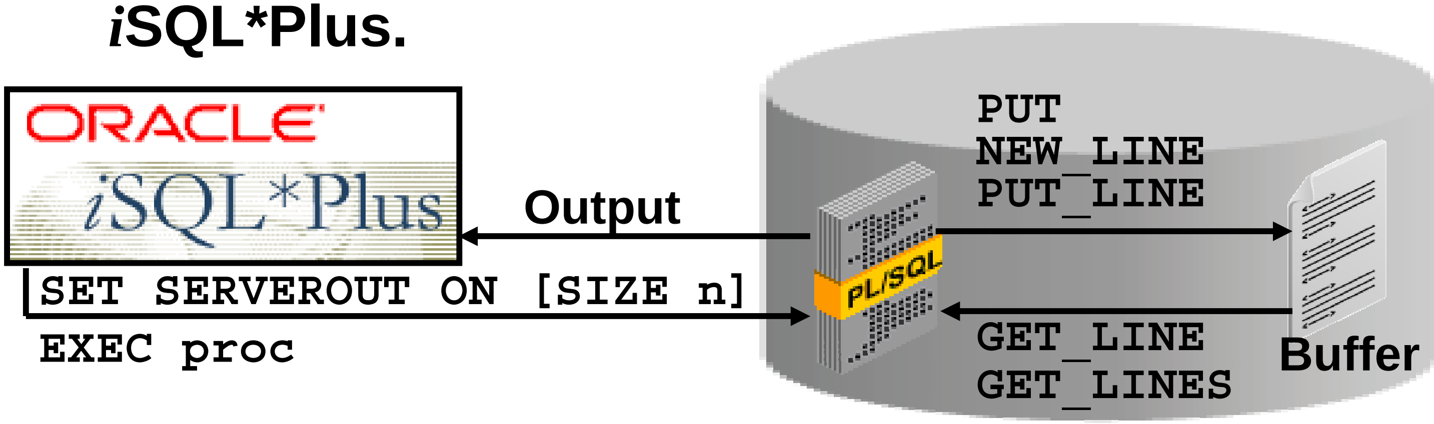
Here is an abbreviated list of some Oracle-supplied packages:

- `DBMS_ALERT`
- `DBMS_LOCK`
- `DBMS_SESSION`
- `DBMS_OUTPUT`
- `HTP`
- `UTL_FILE`
- `UTL_MAIL`
- `DBMS_SCHEDULER`

How the DBMS_OUTPUT Package Works

The DBMS_OUTPUT package enables you to send messages from stored subprograms and triggers.

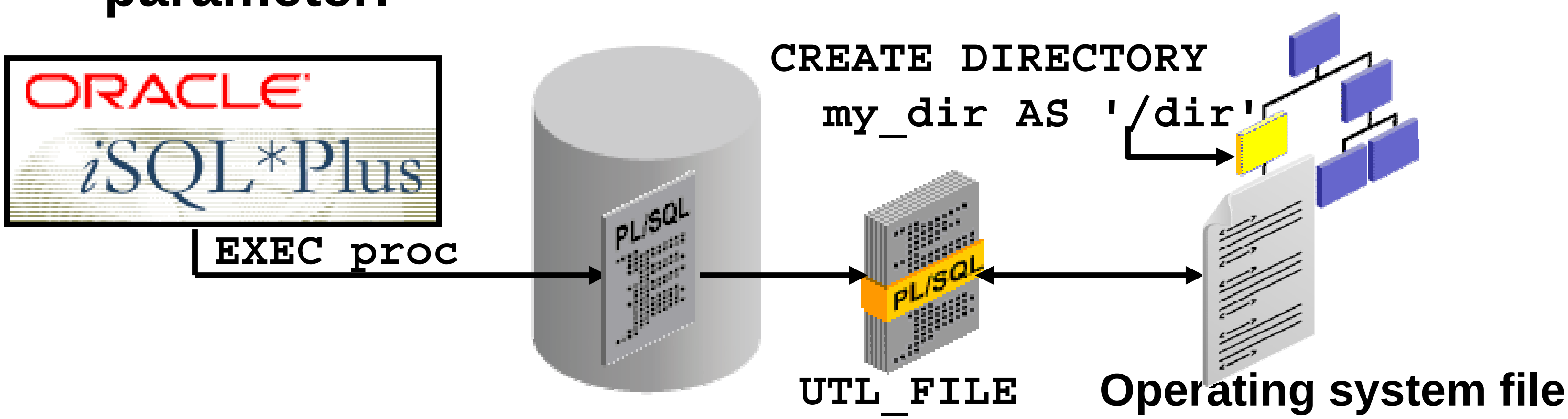
- PUT and PUT_LINE place text in the buffer.
- GET_LINE and GET_LINES read the buffer.
- Messages are not sent until the sender completes.
- Use SET SERVEROUTPUT ON to display messages in *i*SQL*Plus.



Interacting with Operating System Files

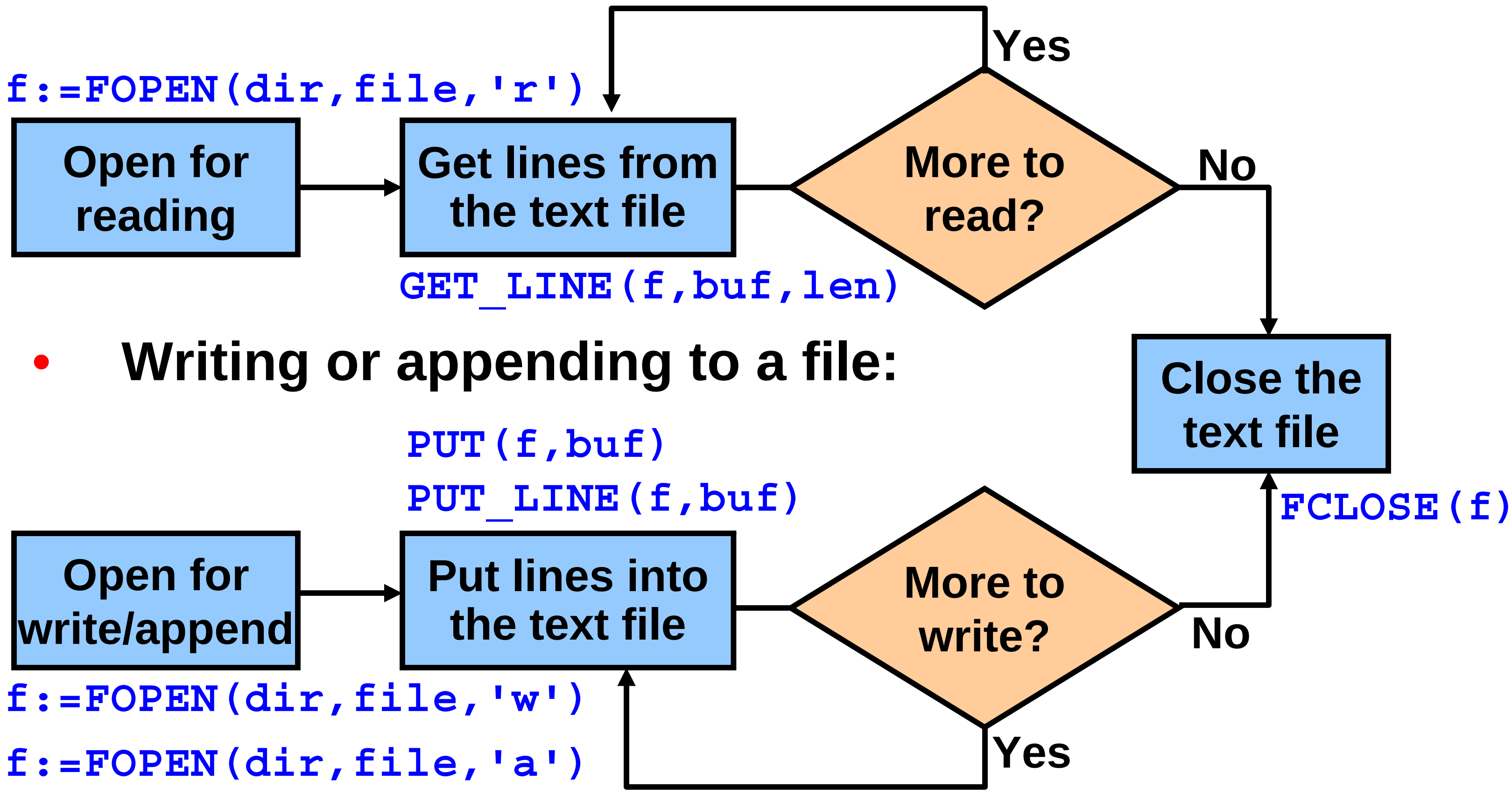
The `UTL_FILE` package extends PL/SQL programs to read and write operating system text files. `UTL_FILE`:

- Provides a restricted version of operating system stream file I/O for text files
- Can access files in operating system directories defined by a `CREATE DIRECTORY` statement. You can also use the `utl_file_dir` database parameter.



File Processing Using the `UTL_FILE` Package

- **Reading a file:**
- **Writing or appending to a file:**



Exceptions in the `UTL_FILE` Package

You may have to handle one of these exceptions when using `UTL_FILE` subprograms:

- `INVALID_PATH`
- `INVALID_MODE`
- `INVALID_FILEHANDLE`
- `INVALID_OPERATION`
- `READ_ERROR`
- `WRITE_ERROR`
- `INTERNAL_ERROR`

Other exceptions not in the `UTL_FILE` package are:

- `NO_DATA_FOUND` and `VALUE_ERROR`

FOPEN and IS_OPEN Function Parameters

```
FUNCTION FOPEN (location IN VARCHAR2,
                filename IN VARCHAR2,
                open_mode IN VARCHAR2)
RETURN UTL_FILE.FILE_TYPE;
```

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

Example:

```
CREATE PROCEDURE read_file(dir VARCHAR2, filename
VARCHAR2) IS file UTL_FILE.FILE_TYPE;
...
BEGIN
  ...
  IF NOT UTL_FILE.IS_OPEN(file) THEN
   file := UTL_FILE.FOPEN (dir, filename, 'R');
   ...
  END IF;
END read_file;
```

ORACLE

Using UTL_FILE: Example

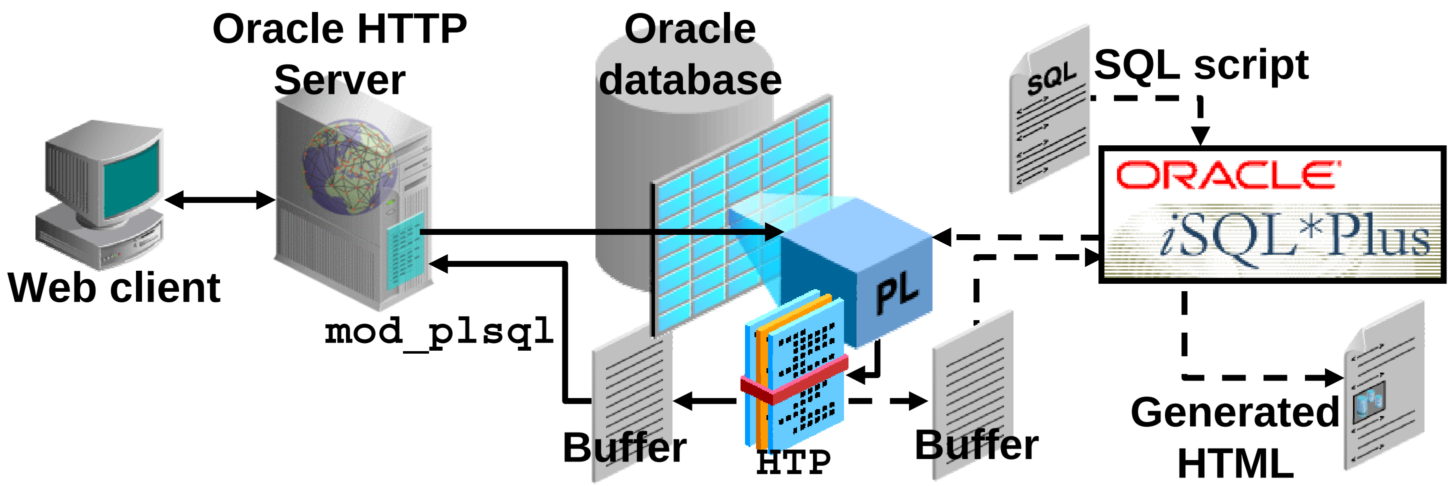
```
CREATE OR REPLACE PROCEDURE sal_status(
  dir IN VARCHAR2, filename IN VARCHAR2) IS
  file UTL_FILE.FILE_TYPE;
 CURSOR empc IS
  SELECT last_name, salary, department_id
  FROM employees ORDER BY department_id;
 newdeptno employees.department_id%TYPE;
 olddeptno employees.department_id%TYPE := 0;
BEGIN
  file:= UTL_FILE.FOPEN (dir, filename, 'w');
  UTL_FILE.PUT_LINE(file,
   'REPORT: GENERATED ON ' || SYSDATE);
  UTL_FILE.NEW_LINE (file); ...
```

Using `UTL_FILE`: Example

```
  FOR emp_rec IN empc LOOP
    IF emp_rec.department_id <> olddeptno THEN
      UTL_FILE.PUT_LINE (file,
       'DEPARTMENT: ' || emp_rec.department_id);
      UTL_FILE.NEW_LINE (file);
    END IF;
    UTL_FILE.PUT_LINE (file,
       '  EMPLOYEE: ' || emp_rec.last_name ||
       ' earns: ' || emp_rec.salary);
    olddeptno := emp_rec.department_id;
    UTL_FILE.NEW_LINE (file);
  END LOOP;
  UTL_FILE.PUT_LINE(file,'*** END OF REPORT ***');
  UTL_FILE.FCLOSE (file);
EXCEPTION
 WHEN UTL_FILE.INVALID_FILEHANDLE THEN
  RAISE_APPLICATION_ERROR(-20001,'Invalid File.');
 WHEN UTL_FILE.WRITE_ERROR THEN
  RAISE_APPLICATION_ERROR (-20002, 'Unable to
write to file');
END sal_status;
/
```

Generating Web Pages with the `HTP` Package

- The `HTP` package procedures generate HTML tags.
- The `HTP` package is used to generate HTML documents dynamically and can be invoked from:
  - A browser using Oracle HTTP Server and PL/SQL Gateway (`mod_plsql`) services
  - An *i*SQL*Plus script to display HTML output



Using the HTP Package Procedures

- Generate one or more HTML tags. For example:

```
htp.bold('Hello');                 -- <B>Hello</B>
htp.print('Hi <B>World</B>');      -- Hi <B>World</B>
```

- Are used to create a well-formed HTML document:

```
BEGIN                                  -- Generates:
 htp.htmlOpen;    --------->           <HTML>
 htp.headOpen;    --------->           <HEAD>
 htp.title('Welcome');  -->            <TITLE>Welcome</TITLE>
 htp.headClose;   --------->           </HEAD>
 htp.bodyOpen;    --------->           <BODY>
 htp.print('My home page');            My home page
 htp.bodyClose;   --------->           </BODY>
 htp.htmlClose;   --------->           </HTML>
END;
```

Creating an HTML File with *i*SQL*Plus

To create an HTML file with *i*SQL*Plus, perform the following steps:

1. Create a SQL script with the following commands:

```
SET SERVEROUTPUT ON
ACCEPT procname PROMPT "Procedure: "
EXECUTE &procname
EXECUTE owa_util.showpage
UNDEFINE proc
```

2. Load and execute the script in *i*SQL*Plus, supplying values for substitution variables.
3. Select, copy, and paste the HTML text that is generated in the browser to an HTML file.
4. Open the HTML file in a browser.

Using UTL_MAIL

The UTL_MAIL package:

- Is a utility for managing e-mail that includes such commonly used e-mail features as attachments, CC, BCC, and return receipt
- Requires the SMTP_OUT_SERVER database initialization parameter to be set
- Provides the following procedures:
  - SEND for messages without attachments
  - SEND_ATTACH_RAW for messages with binary attachments
  - SEND_ATTACH_VARCHAR2 for messages with text attachments

Installing and Using `UTL_MAIL`

- As `SYSDBA`, using *i*SQL*Plus:
  - Set the `SMTP_OUT_SERVER` (requires DBMS restart).

```
ALTER SYSTEM SET SMTP_OUT_SERVER='smtp.server.com'
SCOPE=SPFILE
```

  - Install the `UTL_MAIL` package.

```
@?/rdbms/admin/utlmail.sql
@?/rdbms/admin/prvtmail.plb
```

- As a developer, invoke a `UTL_MAIL` procedure:

```
BEGIN
 UTL_MAIL.SEND('otn@oracle.com','user@oracle.com',
   message => 'For latest downloads visit OTN',
   subject => 'OTN Newsletter');
END;
```

Sending E-Mail with a Binary Attachment

Use the `UTL_MAIL.SEND_ATTACH_RAW` procedure:

```
CREATE OR REPLACE PROCEDURE send_mail_logo IS
BEGIN
  UTL_MAIL.SEND_ATTACH_RAW(
    sender => 'me@oracle.com',
    recipients => 'you@somewhere.net',
    message =>
      '<HTML><BODY>See attachment</BODY></HTML>',
    subject => 'Oracle Logo',
    mime_type => 'text/html'
    attachment => get_image('oracle.gif'),
    att_inline => true,
    att_mime_type => 'image/gif',
    att_filename => 'oralogo.gif');
END;
/
```

Sending E-Mail with a Text Attachment

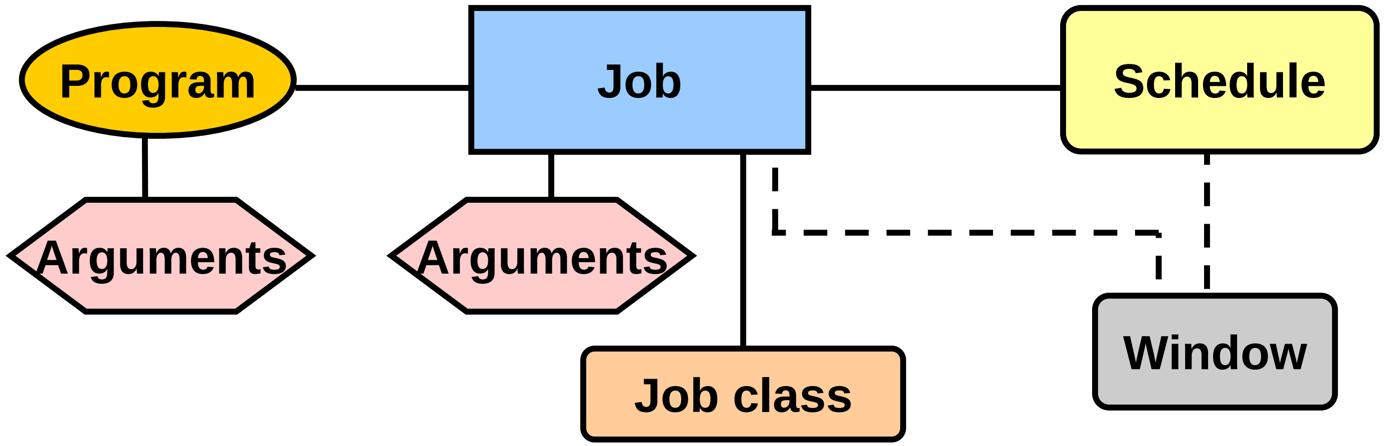
Use the **UTL_MAIL.SEND_ATTACH_VARCHAR2** procedure:

```
CREATE OR REPLACE PROCEDURE send_mail_file IS
BEGIN
  UTL_MAIL.SEND_ATTACH_VARCHAR2(
    sender => 'me@oracle.com',
    recipients => 'you@somewhere.net',
    message =>
      '<HTML><BODY>See attachment</BODY></HTML>',
    subject => 'Oracle Notes',
    mime_type => 'text/html'
    attachment => get_file('notes.txt'),
    att_inline => false,
    att_mime_type => 'text/plain',
    att_filename => 'notes.txt');
END;
/
```

`DBMS_SCHEDULER` Package

The database Scheduler comprises several components to enable jobs to be run. Use the `DBMS_SCHEDULER` package to create each job with:

- A unique job name
- A program (“what” should be executed)
- A schedule (“when” it should run)



Creating a Job

A job can be created in several ways by using a combination of in-line parameters, named `Programs`, and named `Schedules`. You can create a job with the `CREATE_JOB` procedure by:

- Using in-line information with the "what" and the schedule specified as parameters
- Using a named (saved) program and specifying the schedule in-line
- Specifying what should be done in-line and using a named Schedule
- Using named Program and Schedule components

Note: Creating a job requires the `CREATE JOB` system privilege.

ORACLE

Creating a Job with In-Line Parameters

Specify the type of code, code, start time, and frequency of the job to be run in the arguments of the `CREATE_JOB` procedure.

Here is an example that schedules a PL/SQL block every hour:

```
BEGIN
  DBMS_SCHEDULER.CREATE_JOB(
   job_name => 'JOB_NAME',
   job_type => 'PLSQL_BLOCK',
   job_action => 'BEGIN ...; END;',
   start_date => SYSTIMESTAMP,
   repeat_interval=>'FREQUENCY=HOURLY;INTERVAL=1',
   enabled => TRUE);
END;
/
```

Creating a Job Using a Program

- Use `CREATE_PROGRAM` to create a program:

```
BEGIN
  DBMS_SCHEDULER.CREATE_PROGRAM(
   program_name => 'PROG_NAME',
   program_type => 'PLSQL_BLOCK',
   program_action => 'BEGIN ...; END;');
END;
```

- Use overloaded `CREATE_JOB` procedure with its `program_name` parameter:

```
BEGIN
  DBMS_SCHEDULER.CREATE_JOB('JOB_NAME',
   program_name => 'PROG_NAME',
   start_date => SYSTIMESTAMP,
   repeat_interval => 'FREQ=DAILY',
   enabled => TRUE);
END;
```

Creating a Job for a Program with Arguments

- Create a program:

```
DBMS_SCHEDULER.CREATE_PROGRAM(
  program_name => 'PROG_NAME',
  program_type => 'STORED_PROCEDURE',
  program_action => 'EMP_REPORT');
```

- Define an argument:

```
DBMS_SCHEDULER.DEFINE_PROGRAM_ARGUMENT(
  program_name => 'PROG_NAME',
  argument_name => 'DEPT_ID',
  argument_position=> 1, argument_type=> 'NUMBER',
  default_value => '50');
```

- Create a job specifying the number of arguments:

```
DBMS_SCHEDULER.CREATE_JOB('JOB_NAME', program_name
  => 'PROG_NAME', start_date => SYSTIMESTAMP,
  repeat_interval => 'FREQ=DAILY',
  number_of_arguments => 1, enabled => TRUE);
```

Creating a Job Using a Schedule

- Use `CREATE_SCHEDULE` to create a schedule:

```
BEGIN
  DBMS_SCHEDULER.CREATE_SCHEDULE('SCHED_NAME',
   start_date => SYSTIMESTAMP,
   repeat_interval => 'FREQ=DAILY',
   end_date => SYSTIMESTAMP +15);
END;
```

- Use `CREATE_JOB` by referencing the schedule in the `schedule_name` parameter:

```
BEGIN
  DBMS_SCHEDULER.CREATE_JOB('JOB_NAME',
   schedule_name => 'SCHED_NAME',
   job_type => 'PLSQL_BLOCK',
   job_action => 'BEGIN ...; END;',
   enabled => TRUE);
END;
```

Setting the Repeat Interval for a Job

- **Using a calendaring expression:**

```
repeat_interval=> 'FREQ=HOURLY; INTERVAL=4'
repeat_interval=> 'FREQ=DAILY'
repeat_interval=> 'FREQ=MINUTELY;INTERVAL=15'
repeat_interval=> 'FREQ=YEARLY;
                   BYMONTH=MAR,JUN,SEP,DEC;
                   BYMONTHDAY=15'
```

- **Using a PL/SQL expression:**

```
repeat_interval=> 'SYSDATE + 36/24'
repeat_interval=> 'SYSDATE + 1'
repeat_interval=> 'SYSDATE + 15/(24*60)'
```

Creating a Job Using a Named Program and Schedule

- Create a named program called `PROG_NAME` by using the `CREATE_PROGRAM` procedure.
- Create a named schedule called `SCHED_NAME` by using the `CREATE_SCHEDULE` procedure.
- Create a job referencing the named program and schedule:

```
BEGIN
  DBMS_SCHEDULER.CREATE_JOB('JOB_NAME',
   program_name => 'PROG_NAME',
   schedule_name => 'SCHED_NAME',
   enabled => TRUE);
END;
/
```

Managing Jobs

- **Run a job:**

```
DBMS_SCHEDULER.RUN_JOB('SCHEMA.JOB_NAME');
```

- **Stop a job:**

```
DBMS_SCHEDULER.STOP_JOB('SCHEMA.JOB_NAME');
```

- **Drop a job even if it is currently running:**

```
DBMS_SCHEDULER.DROP_JOB('JOB_NAME', TRUE);
```

Data Dictionary Views

- `[DBA | ALL | USER]_SCHEDULER_JOBS`
- `[DBA | ALL | USER]_SCHEDULER_RUNNING_JOBS`
- `[DBA | ALL]_SCHEDULER_JOB_CLASSES`
- `[DBA | ALL | USER]_SCHEDULER_JOB_LOG`
- `[DBA | ALL | USER]_SCHEDULER_JOB_RUN_DETAILS`
- `[DBA | ALL | USER]_SCHEDULER_PROGRAMS`

Summary

In this lesson, you should have learned how to:

- Use various preinstalled packages that are provided by the Oracle server
- Use the following packages:
  - `DBMS_OUTPUT` to buffer and display text
  - `UTL_FILE` to write operating system text files
  - `HTP` to generate HTML documents
  - `UTL_MAIL` to send messages with attachments
  - `DBMS_SCHEDULER` to automate processing
- Create packages individually or by using the `catproc.sql` script

Practice 5: Overview

This practice covers the following topics:

- **Using `UTL_FILE` to generate a text report**
- **Using `HTP` to generate a Web page report**
- **Using `DBMS_SCHEDULER` to automate report processing**

Dynamic SQL and Metadata

Objectives

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

- Describe the execution flow of SQL statements
- Build and execute SQL statements dynamically using Native Dynamic SQL (that is, with `EXECUTE IMMEDIATE` statements)
- Compare Native Dynamic SQL with the `DBMS_SQL` package approach
- Use the `DBMS_METADATA` package to obtain metadata from the data dictionary as XML or creation DDL that can be used to re-create the objects

ORACLE

Execution Flow of SQL

- All SQL statements go through various stages:
  - Parse
  - Bind
  - Execute
  - Fetch
- Some stages may not be relevant for all statements—for example, the fetch phase is applicable to queries.

Note: For embedded SQL statements (`SELECT`, DML, `COMMIT`, and `ROLLBACK`), the parse and bind phases are done at compile time. For dynamic SQL statements, all phases are performed at run time.

Dynamic SQL

Use dynamic SQL to create a SQL statement whose structure may change during run time. Dynamic SQL:

- **Is constructed and stored as a character string within the application**
- **Is a SQL statement with varying column data, or different conditions with or without placeholders (bind variables)**
- **Enables data-definition, data-control, or session-control statements to be written and executed from PL/SQL**
- **Is executed with Native Dynamic SQL statements or the `DBMS_SQL` package**

Native Dynamic SQL

- Provides native support for dynamic SQL directly in the PL/SQL language
- Provides the ability to execute SQL statements whose structure is unknown until execution time
- Is supported by the following PL/SQL statements:
  - `EXECUTE IMMEDIATE`
  - `OPEN-FOR`
  - `FETCH`
  - `CLOSE`

Using the `EXECUTE IMMEDIATE` Statement

Use the `EXECUTE IMMEDIATE` statement for Native Dynamic SQL or PL/SQL anonymous blocks:

```
EXECUTE IMMEDIATE dynamic_string
 [INTO {define_variable
     [, define_variable] ... | record}]
 [USING [IN|OUT|IN OUT] bind_argument
     [, [IN|OUT|IN OUT] bind_argument] ... ];
```

- **`INTO` is used for single-row queries and specifies the variables or records into which column values are retrieved.**
- **`USING` is used to hold all bind arguments. The default parameter mode is `IN`, if not specified.**

Dynamic SQL with a DDL Statement

- **Create a table:**

```
CREATE PROCEDURE create_table(
  table_name VARCHAR2, col_specs  VARCHAR2) IS
BEGIN
  EXECUTE IMMEDIATE 'CREATE TABLE '||table_name||
                    ' (' || col_specs || ')';
END;
/
```

- **Call example:**

```
BEGIN
  create_table('EMPLOYEE_NAMES',
   'id NUMBER(4) PRIMARY KEY, name VARCHAR2(40)');
END;
/
```

Dynamic SQL with DML Statements

- Delete rows from any table:

```
CREATE FUNCTION del_rows(table_name VARCHAR2)
RETURN NUMBER IS
BEGIN
  EXECUTE IMMEDIATE 'DELETE FROM '||table_name;
  RETURN SQL%ROWCOUNT;
END;
```

```
BEGIN DBMS_OUTPUT.PUT_LINE(
  del_rows('EMPLOYEE_NAMES')|| ' rows deleted.');
END;
```

- Insert a row into a table with two columns:

```
CREATE PROCEDURE add_row(table_name VARCHAR2,
   id NUMBER, name VARCHAR2) IS
BEGIN
  EXECUTE IMMEDIATE 'INSERT INTO '||table_name||
        ' VALUES (:1, :2)' USING id, name;
END;
```

ORACLE

Dynamic SQL with a Single-Row Query

Example of a single-row query:

```
CREATE FUNCTION get_emp(emp_id NUMBER)
RETURN employees%ROWTYPE IS
  stmt VARCHAR2(200);
  emprec employees%ROWTYPE;
BEGIN
  stmt := 'SELECT * FROM employees ' ||
          'WHERE employee_id = :id';
  EXECUTE IMMEDIATE stmt INTO emprec USING emp_id;
  RETURN emprec;
END;
/
```

```
DECLARE
  emprec employees%ROWTYPE := get_emp(100);
BEGIN
  DBMS_OUTPUT.PUT_LINE('Emp: '||emprec.last_name);
END;
/
```

Dynamic SQL with a Multirow Query

Use `OPEN-FOR`, `FETCH`, and `CLOSE` processing:

```
CREATE PROCEDURE list_employees(deptid NUMBER) IS
  TYPE emp_refcsr IS REF CURSOR;
  emp_cv  emp_refcsr;
  emprec  employees%ROWTYPE;
  stmt varchar2(200) := 'SELECT * FROM employees';
BEGIN
  IF deptid IS NULL THEN OPEN emp_cv FOR stmt;
  ELSE
    stmt := stmt || ' WHERE department_id = :id';
    OPEN emp_cv FOR stmt USING deptid;
  END IF;
  LOOP
    FETCH emp_cv INTO emprec;
    EXIT WHEN emp_cv%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(emprec.department_id||
                         ' ' ||emprec.last_name);
  END LOOP;
  CLOSE emp_cv;
END;
```

Declaring Cursor Variables

- Declare a cursor type as `REF CURSOR`:

```
CREATE PROCEDURE process_data IS
  TYPE ref_ctype IS REF CURSOR; -- weak ref cursor
  TYPE emp_ref_ctype IS REF CURSOR -- strong
     RETURN employees%ROWTYPE;
  :
```

- Declare a cursor variable using the cursor type:

```
  :
  dept_csrvar ref_ctype;
  emp_csrvar  emp_ref_ctype;
BEGIN
  OPEN emp_csrvar FOR SELECT * FROM employees;
  OPEN dept_csrvar FOR SELECT * from departments;
  -- Then use as normal cursors
END;
```

Dynamically Executing a PL/SQL Block

Execute a PL/SQL anonymous block dynamically:

```
CREATE FUNCTION annual_sal(emp_id NUMBER)
RETURN NUMBER IS
  plsql varchar2(200) :=
    'DECLARE '||
    ' emprec employees%ROWTYPE; '||
    'BEGIN '||
    ' emprec := get_emp(:empid); ' ||
    ' :res := emprec.salary * 12; ' ||
    'END;';
  result NUMBER;
BEGIN
 EXECUTE IMMEDIATE plsql
         USING IN emp_id, OUT result;
  RETURN result;
END;
/
```

```
EXECUTE DBMS_OUTPUT.PUT_LINE(annual_sal(100))
```

Using Native Dynamic SQL to Compile PL/SQL Code

Compile PL/SQL code with the `ALTER` statement:

- `ALTER PROCEDURE name COMPILE`
- `ALTER FUNCTION name COMPILE`
- `ALTER PACKAGE name COMPILE SPECIFICATION`
- `ALTER PACKAGE name COMPILE BODY`

```
CREATE PROCEDURE compile_plsql(name VARCHAR2,
 plsql_type VARCHAR2, options VARCHAR2 := NULL) IS
  stmt varchar2(200) := 'ALTER '|| plsql_type ||
                        ' '|| name || ' COMPILE';
BEGIN
 IF options IS NOT NULL THEN
   stmt := stmt || ' ' || options;
 END IF;
 EXECUTE IMMEDIATE stmt;
END;
/
```

Using the `DBMS_SQL` Package

The `DBMS_SQL` package is used to write dynamic SQL in stored procedures and to parse DDL statements. Some of the procedures and functions of the package include:

- `OPEN_CURSOR`
- `PARSE`
- `BIND_VARIABLE`
- `EXECUTE`
- `FETCH_ROWS`
- `CLOSE_CURSOR`

Using `DBMS_SQL` with a DML Statement

Example of deleting rows:

```
CREATE OR REPLACE FUNCTION delete_all_rows
  (table_name VARCHAR2) RETURN NUMBER IS
  csr_id INTEGER;
  rows_del    NUMBER;
BEGIN
  csr_id := DBMS_SQL.OPEN_CURSOR;
  DBMS_SQL.PARSE(csr_id,
    'DELETE FROM '||table_name, DBMS_SQL.NATIVE);
  rows_del := DBMS_SQL.EXECUTE (csr_id);
  DBMS_SQL.CLOSE_CURSOR(csr_id);
  RETURN rows_del;
END;
/
```

```
CREATE table temp_emp as select * from employees;
BEGIN
 DBMS_OUTPUT.PUT_LINE('Rows Deleted: ' ||
delete_all_rows('temp_emp'));
END;
/
```

Using `DBMS_SQL` with a Parameterized DML Statement

```
CREATE PROCEDURE insert_row (table_name VARCHAR2,
 id VARCHAR2, name VARCHAR2, region NUMBER) IS
  csr_id     INTEGER;
  stmt       VARCHAR2(200);
  rows_added NUMBER;
BEGIN
  stmt := 'INSERT INTO '||table_name||
          ' VALUES (:cid, :cname, :rid)';
  csr_id := DBMS_SQL.OPEN_CURSOR;
  DBMS_SQL.PARSE(csr_id, stmt, DBMS_SQL.NATIVE);
  DBMS_SQL.BIND_VARIABLE(csr_id, ':cid', id);
  DBMS_SQL.BIND_VARIABLE(csr_id, ':cname', name);
  DBMS_SQL.BIND_VARIABLE(csr_id, ':rid', region);
  rows_added := DBMS_SQL.EXECUTE(csr_id);
  DBMS_SQL.CLOSE_CURSOR(csr_id);
  DBMS_OUTPUT.PUT_LINE(rows_added||' row added');
END;
/
```

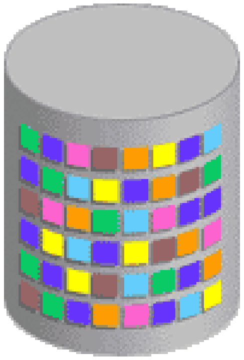
Comparison of Native Dynamic SQL and the `DBMS_SQL` Package

Native Dynamic SQL:

- **Is easier to use than `DBMS_SQL`**
- **Requires less code than `DBMS_SQL`**
- **Enhances performance because the PL/SQL interpreter provides native support for it**
- **Supports all types supported by static SQL in PL/SQL, including user-defined types**
- **Can fetch rows directly into PL/SQL records**

DBMS_METADATA Package

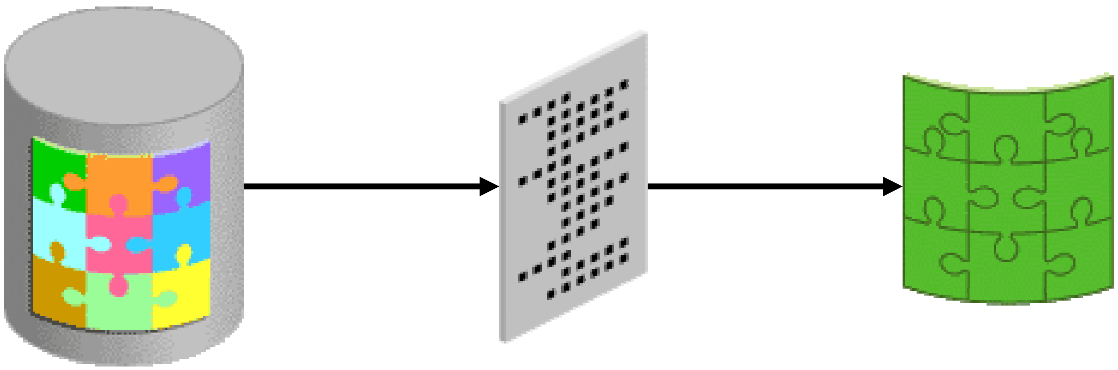
The DBMS_METADATA package provides a centralized facility for the extraction, manipulation, and resubmission of dictionary metadata.



Metadata API

Processing involves the following steps:

1. Fetch an object's metadata as XML.
2. Transform the XML in a variety of ways (including transforming it into SQL DDL).
3. Submit the XML to re-create the object.



Subprograms in DBMS_METADATA

Name	Description
OPEN	Specifies the type of object to be retrieved, the version of its metadata, and the object model. The return value is an opaque context handle for the set of objects.
SET_FILTER	Specifies restrictions on the objects to be retrieved such as the object name or schema
SET_COUNT	Specifies the maximum number of objects to be retrieved in a single FETCH_xxx call
GET_QUERY	Returns the text of the queries that will be used by FETCH_xxx
SET_PARSE_ITEM	Enables output parsing and specifies an object attribute to be parsed and returned
ADD_TRANSFORM	Specifies a transform that FETCH_xxx applies to the XML representation of the retrieved objects
SET_TRANSFORM_PARAM, SET_REMAP_PARAM	Specifies parameters to the XSLT stylesheet identified by transform_handle
FETCH_*XXX*	Returns metadata for objects meeting the criteria established by OPEN, SET_FILTER
CLOSE	Invalidates the handle returned by OPEN and cleans up the associated state

FETCH_xxx Subprograms

Name	Description
FETCH_XML	This function returns the XML metadata for an object as an XMLType.
FETCH_DDL	This function returns the DDL (either to create or to drop the object) into a predefined nested table.
FETCH_CLOB	This function returns the objects (transformed or not) as a CLOB.
FETCH_XML_CLOB	This procedure returns the XML metadata for the objects as a CLOB in an IN OUT NOCOPY parameter to avoid expensive LOB copies.

`SET_FILTER` Procedure

- **Syntax:**

```
PROCEDURE set_filter
( handle IN NUMBER,
  name   IN VARCHAR2,
  value  IN VARCHAR2|BOOLEAN|NUMBER,
  object_type_path VARCHAR2
);
```

- **Example:**

```
...
DBMS_METADATA.SET_FILTER (handle, 'NAME',
'HR');
...
```

Filters

There are over 70 filters, which are organized into object type categories such as:

- Named objects
- Tables
- Objects dependent on tables
- Index
- Dependent objects
- Granted objects
- Table data
- Index statistics
- Constraints
- All object types
- Database export

Examples of Setting Filters

Set up the filter to fetch the `HR` schema objects excluding the object types of functions, procedures, and packages, as well as any views that contain `PAYROLL` in the start of the view name:

```
DBMS_METADATA.SET_FILTER(handle, 'SCHEMA_EXPR',
  'IN (''PAYROLL'', ''HR'')');
DBMS_METADATA.SET_FILTER(handle, 'EXCLUDE_PATH_EXPR',
  '=''FUNCTION''');
DBMS_METADATA.SET_FILTER(handle, 'EXCLUDE_PATH_EXPR',
  '=''PROCEDURE''');
DBMS_METADATA.SET_FILTER(handle, 'EXCLUDE_PATH_EXPR',
  '=''PACKAGE''');
DBMS_METADATA.SET_FILTER(handle, 'EXCLUDE_NAME_EXPR',
  'LIKE ''PAYROLL%''', 'VIEW');
```

ORACLE

Programmatic Use: Example 1

```
CREATE PROCEDURE example_one IS
  h     NUMBER; th1  NUMBER; th2  NUMBER;
  doc   sys.ku$_ddls;
BEGIN
  h := DBMS_METADATA.OPEN('SCHEMA_EXPORT');
  DBMS_METADATA.SET_FILTER (h,'SCHEMA','HR');
  th1  := DBMS_METADATA.ADD_TRANSFORM (h,
    'MODIFY', NULL, 'TABLE');
  DBMS_METADATA.SET_REMAP_PARAM(th1,
    'REMAP_TABLESPACE', 'SYSTEM', 'TBS1');
  th2  :=DBMS_METADATA.ADD_TRANSFORM(h, 'DDL');
  DBMS_METADATA.SET_TRANSFORM_PARAM(th2,
    'SQLTERMINATOR', TRUE);
  DBMS_METADATA.SET_TRANSFORM_PARAM(th2,
    'REF_CONSTRAINTS', FALSE, 'TABLE');
  LOOP
    doc := DBMS_METADATA.FETCH_DDL(h);
    EXIT WHEN doc IS NULL;
  END LOOP;
  DBMS_METADATA.CLOSE(h);
END;
```

Programmatic Use: Example 2

```
CREATE FUNCTION get_table_md RETURN CLOB IS
 h    NUMBER; -- returned by 'OPEN'
 th   NUMBER; -- returned by 'ADD_TRANSFORM'
 doc  CLOB;
BEGIN
 -- specify the OBJECT TYPE
 h := DBMS_METADATA.OPEN('TABLE');
 -- use FILTERS to specify the objects desired
 DBMS_METADATA.SET_FILTER(h,'SCHEMA','HR');
 DBMS_METADATA.SET_FILTER(h,'NAME','EMPLOYEES');
 -- request to be TRANSFORMED into creation DDL
 th := DBMS_METADATA.ADD_TRANSFORM(h,'DDL');
 -- FETCH the object
 doc := DBMS_METADATA.FETCH_CLOB(h);
 -- release resources
 DBMS_METADATA.CLOSE(h);
 RETURN doc;
END;
/
```

Browsing APIs

Name	Description
`GET_`*`XXX`*	The `GET_XML` and `GET_DDL` functions return metadata for a single named object.
`GET_DEPENDENT_`*`XXX`*	This function returns metadata for a dependent object.
`GET_GRANTED_`*`XXX`*	This function returns metadata for a granted object.

Where *`xxx`* is:	`DDL` or `XML`

Browsing APIs: Examples

1. **Get the XML representation of `HR.EMPLOYEES`:**

```
SELECT DBMS_METADATA.GET_XML
                     ('TABLE', 'EMPLOYEES', 'HR')
FROM   dual;
```

2. **Fetch the DDL for all object grants on `HR.EMPLOYEES`:**

```
SELECT DBMS_METADATA.GET_DEPENDENT_DDL
                    ('OBJECT_GRANT', 'EMPLOYEES', 'HR')
FROM   dual;
```

3. **Fetch the DDL for all system grants granted to `HR`:**

```
SELECT DBMS_METADATA.GET_GRANTED_DDL
                 ('SYSTEM_GRANT', 'HR')
FROM   dual;
```

Browsing APIs: Examples

```
BEGIN
 DBMS_METADATA.SET_TRANSFORM_PARAM(
   DBMS_METADATA.SESSION_TRANSFORM,
   'STORAGE', false);
END;
/
SELECT DBMS_METADATA.GET_DDL('TABLE',u.table_name)
FROM   user_all_tables u
WHERE  u.nested = 'NO'
AND    (u.iot_type IS NULL OR u.iot_type = 'IOT');

BEGIN
 DBMS_METADATA.SET_TRANSFORM_PARAM(
    DBMS_METADATA.SESSION_TRANSFORM, 'DEFAULT'):
END;
/
```

1

2

3

Summary

In this lesson, you should have learned how to:

- Explain the execution flow of SQL statements
- Create SQL statements dynamically and execute them using either Native Dynamic SQL statements or the `DBMS_SQL` package
- Recognize the advantages of using Native Dynamic SQL compared to the `DBMS_SQL` package
- Use `DBMS_METADATA` subprograms to programmatically obtain metadata from the data dictionary

Practice 6: Overview

This practice covers the following topics:

- **Creating a package that uses Native Dynamic SQL to create or drop a table and to populate, modify, and delete rows from a table**
- **Creating a package that compiles the PL/SQL code in your schema**
- **Using `DBMS_METADATA` to display the statement to regenerate a PL/SQL subprogram**

7

Design Considerations for PL/SQL Code

Objectives

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

- Use package specifications to create standard constants and exceptions
- Write and call local subprograms
- Set the `AUTHID` directive to control the run-time privileges of a subprogram
- Execute subprograms to perform autonomous transactions
- Use bulk binding and the `RETURNING` clause with DML
- Pass parameters by reference using a `NOCOPY` hint
- Use the `PARALLEL ENABLE` hint for optimization

Standardizing Constants and Exceptions

Constants and exceptions are typically implemented using a bodiless package (that is, in a package specification).

- **Standardizing helps to:**
  - **Develop programs that are consistent**
  - **Promote a higher degree of code reuse**
  - **Ease code maintenance**
  - **Implement company standards across entire applications**
- **Start with standardization of:**
  - **Exception names**
  - **Constant definitions**

Standardizing Exceptions

Create a standardized error-handling package that includes all named and programmer-defined exceptions to be used in the application.

```
CREATE OR REPLACE PACKAGE error_pkg IS
  fk_err        EXCEPTION;
  seq_nbr_err   EXCEPTION;
  PRAGMA EXCEPTION_INIT (fk_err, -2292);
  PRAGMA EXCEPTION_INIT (seq_nbr_err, -2277);
  ...
END error_pkg;
/
```

Standardizing Exception Handling

Consider writing a subprogram for common exception handling to:

- **Display errors based on `SQLCODE` and `SQLERRM` values for exceptions**
- **Track run-time errors easily by using parameters in your code to identify:**
  - **The procedure in which the error occurred**
  - **The location (line number) of the error**
  - **`RAISE_APPLICATION_ERROR` using stack trace capabilities, with the third argument set to `TRUE`**

Standardizing Constants

For programs that use local variables whose values should not change:

- Convert the variables to constants to reduce maintenance and debugging
- Create one central package specification and place all constants in it

```
CREATE OR REPLACE PACKAGE constant_pkg IS
  c_order_received CONSTANT VARCHAR(2) := 'OR';
  c_order_shipped  CONSTANT VARCHAR(2) := 'OS';
  c_min_sal        CONSTANT NUMBER(3)  := 900;
  ...
END constant_pkg;
```

Local Subprograms

- A local subprogram is a `PROCEDURE` or `FUNCTION` defined in the declarative section.

```
CREATE PROCEDURE employee_sal(id NUMBER) IS
   emp employees%ROWTYPE;
   FUNCTION tax(salary VARCHAR2) RETURN NUMBER IS
   BEGIN
     RETURN salary * 0.825;
   END tax;
BEGIN
   SELECT * INTO emp
   FROM EMPLOYEES WHERE employee_id = id;
   DBMS_OUTPUT.PUT_LINE('Tax: '||tax(emp.salary));
END;
```

- The local subprogram must be defined at the end of the declarative section.

Definer's Rights Versus Invoker's Rights

Definer's rights:

- **Used prior to Oracle8*i***
- **Programs execute with the privileges of the creating user.**
- **User does not require privileges on underlying objects that the procedure accesses. User requires privilege only to execute a procedure.**

Invoker's rights:

- **Introduced in Oracle8*i***
- **Programs execute with the privileges of the calling user.**
- **User requires privileges on the underlying objects that the procedure accesses.**

Specifying Invoker's Rights

Set `AUTHID` to `CURRENT_USER`:

```
CREATE OR REPLACE PROCEDURE add_dept(
  id NUMBER, name VARCHAR2) AUTHID CURRENT_USER IS
BEGIN
  INSERT INTO departments
  VALUES (id,name,NULL,NULL);
END;
```

When used with stand-alone functions, procedures, or packages:

- **Names used in queries, DML, Native Dynamic SQL, and `DBMS_SQL` package are resolved in the invoker's schema**
- **Calls to other packages, functions, and procedures are resolved in the definer's schema**

Autonomous Transactions

- Are independent transactions started by another main transaction.
- Are specified with `PRAGMA AUTONOMOUS_TRANSACTION`

```
    PROCEDURE proc1 IS
      emp_id  NUMBER;
    BEGIN
      emp_id := 1234;
      COMMIT;
1     INSERT ...
2     proc2;
      DELETE ...
7     COMMIT;
    END proc1;
```

```
    PROCEDURE proc2 IS
     PRAGMA
      AUTONOMOUS_TRANSACTION;
      dept_id  NUMBER := 90;
3    BEGIN
4      UPDATE ...
       INSERT ...
5      COMMIT;    -- Required
6    END proc2;
```

Features of Autonomous Transactions

Autonomous transactions:

- Are independent of the main transaction
- Suspend the calling transaction until it is completed
- Are not nested transactions
- Do not roll back if the main transaction rolls back
- Enable the changes to become visible to other transactions upon a commit
- Are demarcated (started and ended) by individual subprograms and not by nested or anonymous PL/SQL blocks

ORACLE

Using Autonomous Transactions

Example:

```
PROCEDURE bank_trans(cardnbr NUMBER,loc NUMBER) IS
BEGIN
  log_usage (cardnbr, loc);
  INSERT INTO txn VALUES (9001, 1000,...);
END bank_trans;
```

```
PROCEDURE log_usage (card_id NUMBER, loc NUMBER)
IS
  PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
  INSERT INTO usage
  VALUES (card_id, loc);
  COMMIT;
END log_usage;
```

RETURNING Clause

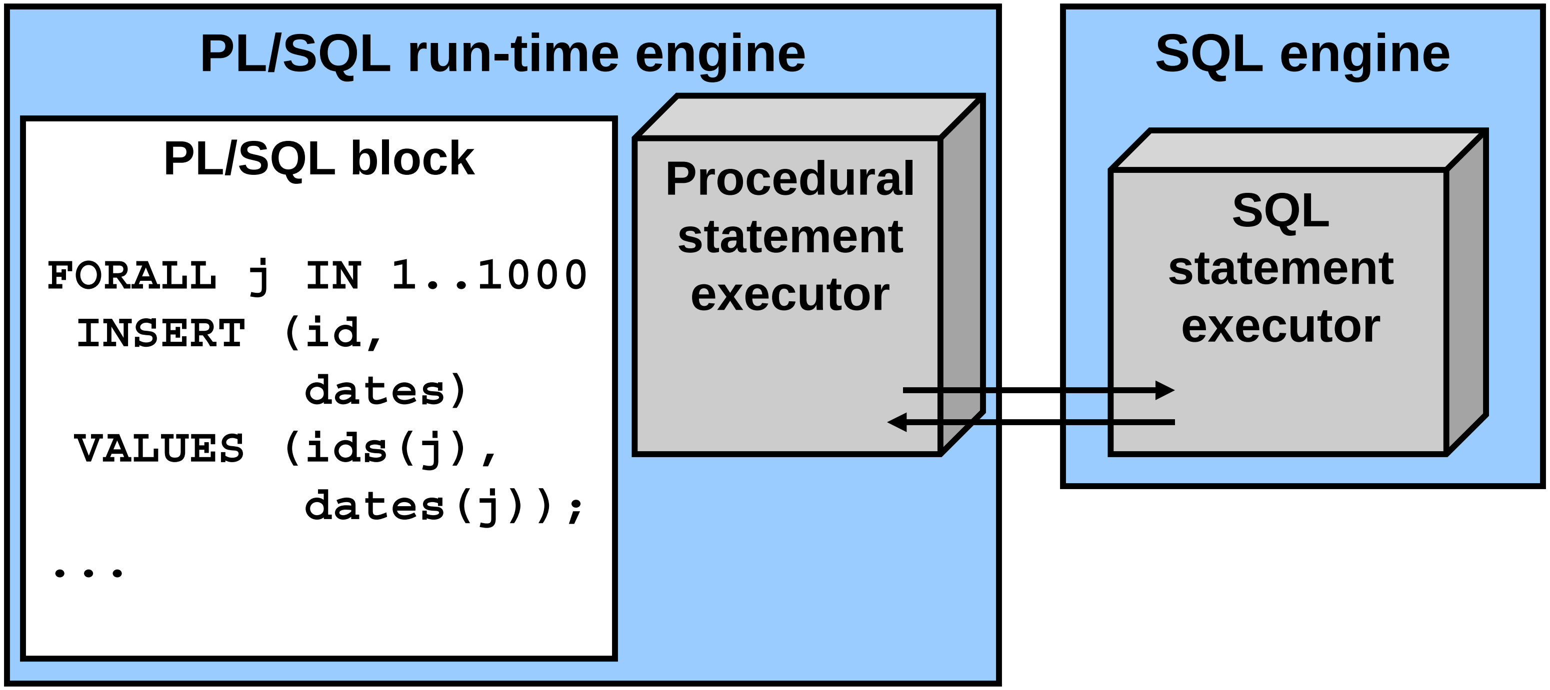
The RETURNING clause:

- Improves performance by returning column values with INSERT, UPDATE, and DELETE statements
- Eliminates the need for a SELECT statement

```
CREATE PROCEDURE update_salary(emp_id NUMBER) IS
  name    employees.last_name%TYPE;
  new_sal employees.salary%TYPE;
BEGIN
  UPDATE employees
    SET salary = salary * 1.1
  WHERE employee_id = emp_id
  RETURNING last_name, salary INTO name, new_sal;
END update_salary;
/
```

Bulk Binding

Binds whole arrays of values in a single operation, rather than using a loop to perform a `FETCH`, `INSERT`, `UPDATE`, and `DELETE` operation multiple times



Using Bulk Binding

Keywords to support bulk binding:

- The `FORALL` keyword instructs the PL/SQL engine to bulk bind input collections before sending them to the SQL engine.

```
FORALL index IN lower_bound .. upper_bound
  [SAVE EXCEPTIONS]
  sql_statement;
```

- The `BULK COLLECT` keyword instructs the SQL engine to bulk bind output collections before returning them to the PL/SQL engine.

```
... BULK COLLECT INTO
      collection_name[,collection_name] ...
```

Bulk Binding `FORALL`: Example

```
CREATE PROCEDURE raise_salary(percent NUMBER) IS
  TYPE numlist IS TABLE OF NUMBER
    INDEX BY BINARY_INTEGER;
  id  numlist;
BEGIN
 id(1) := 100; id(2) := 102;
 id(3) := 104; id(4) := 110;
 -- bulk-bind the PL/SQL table
  FORALL i IN id.FIRST .. id.LAST
    UPDATE employees
      SET salary = (1 + percent/100) * salary
      WHERE manager_id = id(i);
END;
/
```

```
EXECUTE raise_salary(10)
```

Using BULK COLLECT INTO with Queries

The SELECT statement has been enhanced to support the BULK COLLECT INTO syntax.

Example:

```
CREATE PROCEDURE get_departments(loc NUMBER) IS
  TYPE dept_tabtype IS
    TABLE OF departments%ROWTYPE;
  depts dept_tabtype;
BEGIN
  SELECT * BULK COLLECT INTO depts
  FROM departments
  WHERE location_id = loc;
  FOR I IN 1 .. depts.COUNT LOOP
    DBMS_OUTPUT.PUT_LINE(depts(i).department_id
      ||' '||depts(i).department_name);
  END LOOP;
END;
```

Using BULK COLLECT INTO with Cursors

The FETCH statement has been enhanced to support the BULK COLLECT INTO syntax.

Example:

```
CREATE PROCEDURE get_departments(loc NUMBER) IS
  CURSOR dept_csr IS SELECT * FROM departments
                     WHERE location_id = loc;
  TYPE dept_tabtype IS TABLE OF dept_csr%ROWTYPE;
  depts dept_tabtype;
BEGIN
  OPEN dept_csr;
  FETCH dept_csr BULK COLLECT INTO depts;
  CLOSE dept_csr;
 FOR I IN 1 .. depts.COUNT LOOP
    DBMS_OUTPUT.PUT_LINE(depts(i).department_id
     ||' '||depts(i).department_name);
  END LOOP;
END;
```

Using BULK COLLECT INTO with a RETURNING Clause

Example:

```
CREATE PROCEDURE raise_salary(rate NUMBER) IS
   TYPE emplist IS TABLE OF NUMBER;
   TYPE numlist IS TABLE OF employees.salary%TYPE
     INDEX BY BINARY_INTEGER;
   emp_ids  emplist := emplist(100,101,102,104);
   new_sals numlist;
BEGIN
  FORALL i IN emp_ids.FIRST .. emp_ids.LAST
    UPDATE employees
      SET commission_pct = rate * salary
    WHERE employee_id = emp_ids(i)
    RETURNING salary BULK COLLECT INTO new_sals;
  FOR i IN 1 .. new_sals.COUNT LOOP ...
END;
```

Using the `NOCOPY` Hint

The `NOCOPY` hint:

- Is a request to the PL/SQL compiler to pass `OUT` and `IN OUT` parameters by reference rather than by value
- Enhances performance by reducing overhead when passing parameters

```
DECLARE
  TYPE emptabtype IS TABLE OF employees%ROWTYPE;
  emp_tab emptabtype;
  PROCEDURE populate(tab IN OUT NOCOPY emptabtype)
  IS BEGIN ... END;
BEGIN
  populate(emp_tab);
END;
/
```

Effects of the `NOCOPY` Hint

- If the subprogram exits with an exception that is not handled:
  - You cannot rely on the values of the actual parameters passed to a `NOCOPY` parameter
  - Any incomplete modifications are not “rolled back”
- The remote procedure call (RPC) protocol enables you to pass parameters only by value.

`NOCOPY` Hint Can Be Ignored

The `NOCOPY` hint has no effect if:

- The actual parameter:
  - Is an element of an index-by table
  - Is constrained (for example, by scale or `NOT NULL`)
  - And formal parameter are records, where one or both records were declared by using `%ROWTYPE` or `%TYPE`, and constraints on corresponding fields in the records differ
  - Requires an implicit data type conversion
- The subprogram is involved in an external or remote procedure call

`PARALLEL_ENABLE` Hint

The `PARALLEL_ENABLE` hint:

- **Can be used in functions as an optimization hint**

```
CREATE OR REPLACE FUNCTION f2 (p1 NUMBER)
  RETURN NUMBER PARALLEL_ENABLE IS
BEGIN
  RETURN p1 * 2;
END f2;
```

- **Indicates that a function can be used in a parallelized query or parallelized DML statement**

Summary

In this lesson, you should have learned how to:

- Create standardized constants and exceptions using packages
- Develop and invoke local subprograms
- Control the run-time privileges of a subprogram by setting the `AUTHID` directive
- Execute autonomous transactions
- Use the `RETURNING` clause with DML statements, and bulk binding collections with the `FORALL` and `BULK COLLECT INTO` clauses
- Pass parameters by reference using a `NOCOPY` hint
- Enable optimization with `PARALLEL ENABLE` hints

Practice 7: Overview

This practice covers the following topics:

- Creating a package that uses bulk fetch operations
- Creating a local subprogram to perform an autonomous transaction to audit a business operation
- Testing `AUTHID` functionality

8

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 on stored procedures and functions
- Manage procedural dependencies

Understanding Dependencies

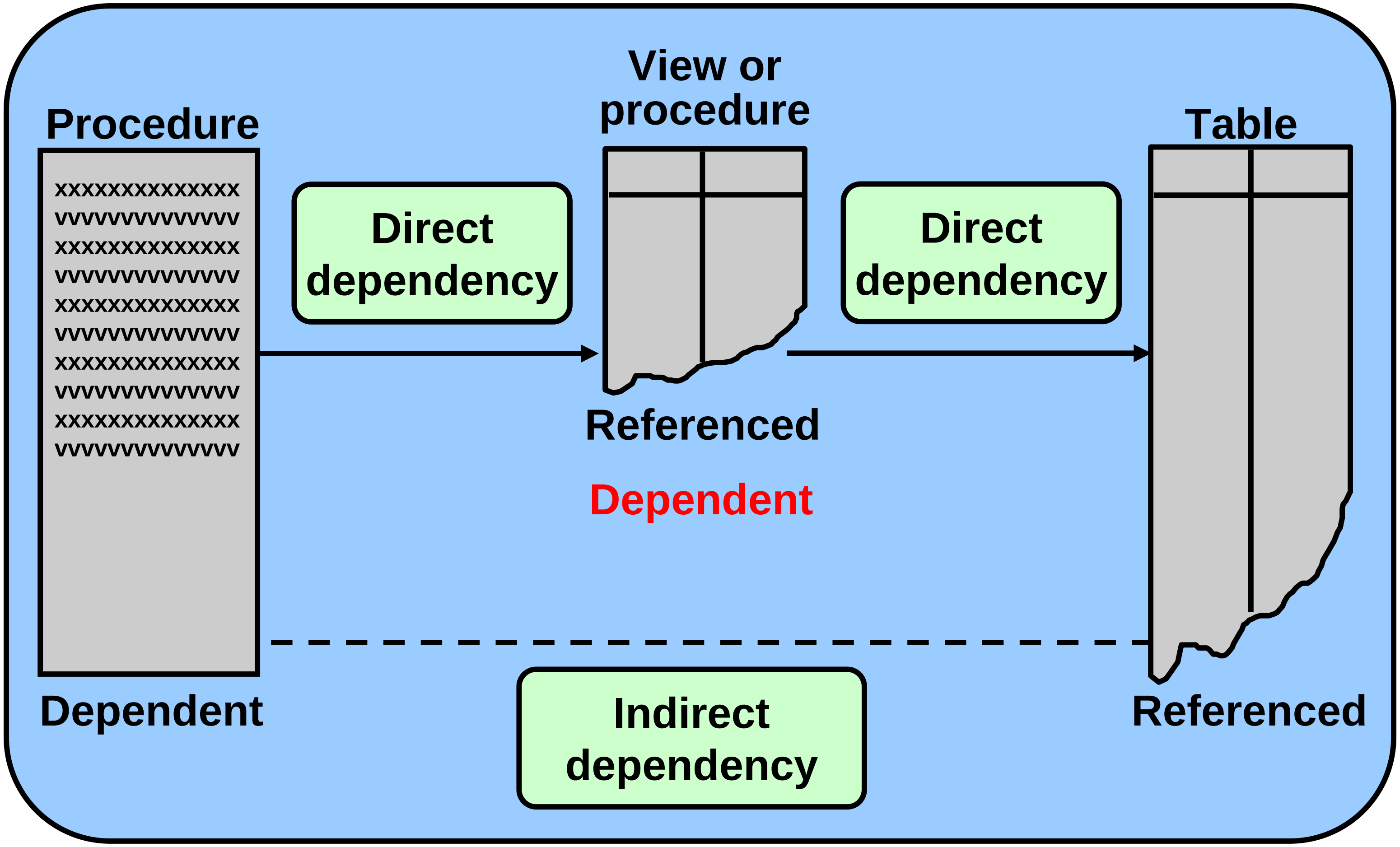
Dependent objects

- Database trigger
- Function
- Procedure
- Package body
- Package specification
- User-defined object and collection types
- View

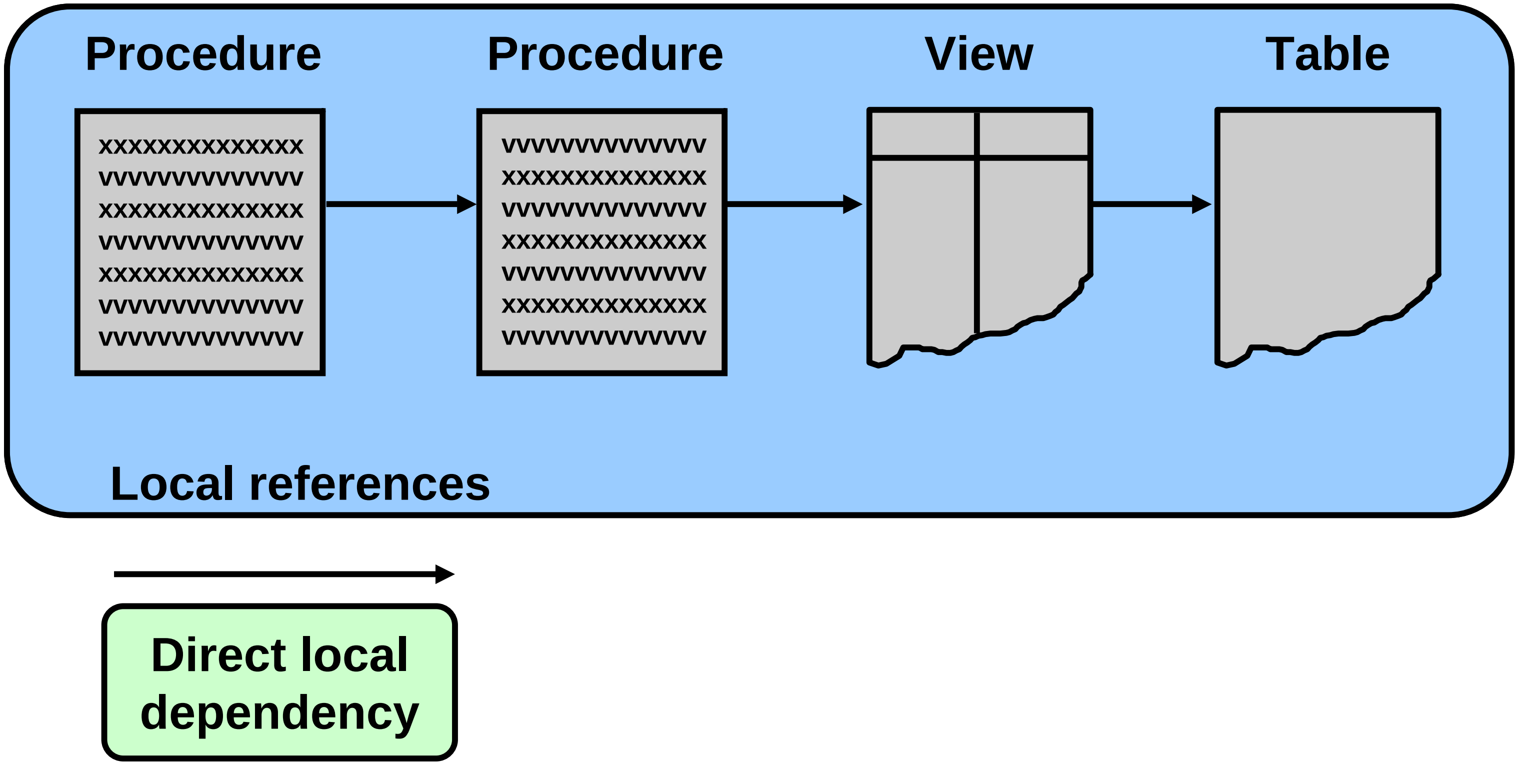
Referenced objects

- Function
- Package specification
- Procedure
- Sequence
- Synonym
- Table
- User-defined object and collection types
- View

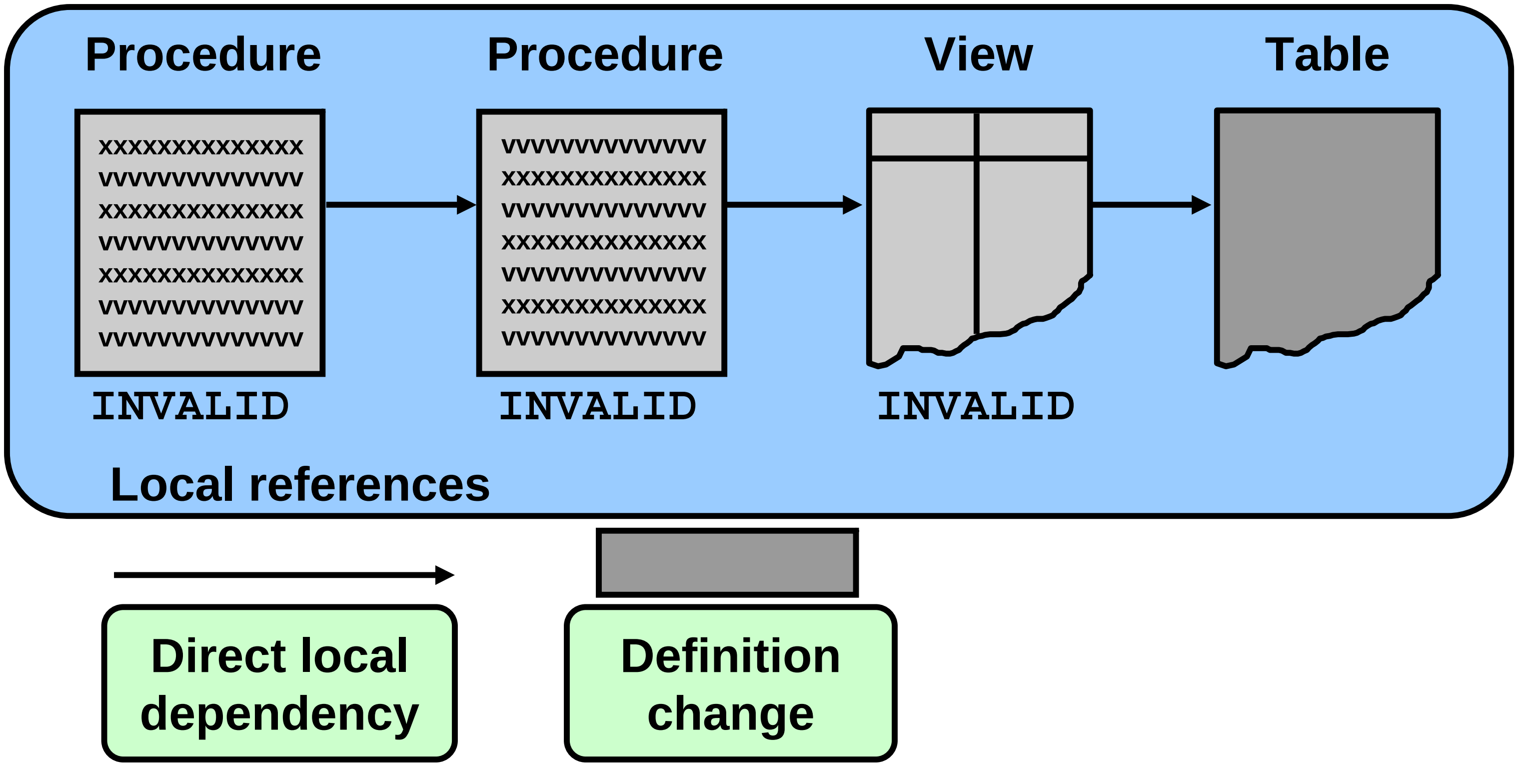
Dependencies



Local Dependencies



Local Dependencies



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

A Scenario of Local Dependencies

ADD_EMP procedure

xxxxxxxxxxxxxxxxxxxxx
vvvvvvvvvvvvvvvvvvvvv
vvvvvvvvvvvvvvvv
vvvvvvvvvvvvvvvvvvvvv
vvvvvvvvvvvvvvvvvvvvv
vvvvvxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxx
vvvvvvvvvvvvvvvvvvvvv

EMP_VW view

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

...

QUERY_EMP procedure

xxxxxxxxxxxxxxxxxxxxx
vvvvvvvvvvvvvvvvvvvvv
vvvvvvvvvvvvvvvv
vvvvvvvvvvvvvvvvvvvvv
vvvvvvvvvvvvvvvvvvvvv
vvvvvxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxx
vvvvvvvvvvvvvvvvvvvvv

EMPLOYEES table

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NI
100	Steven	King	SKING	515.123.4567
101	Neena	Kochhar	NKOCHHAR	515.123.4568
102	Lex	De Haan	LDEHAAN	515.123.4569
105	David	Austin	DAUSTIN	590.423.4569
108	Nancy	Greenberg	NGREENBE	515.124.4569

...

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 `utldtree.sql` script 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')
```

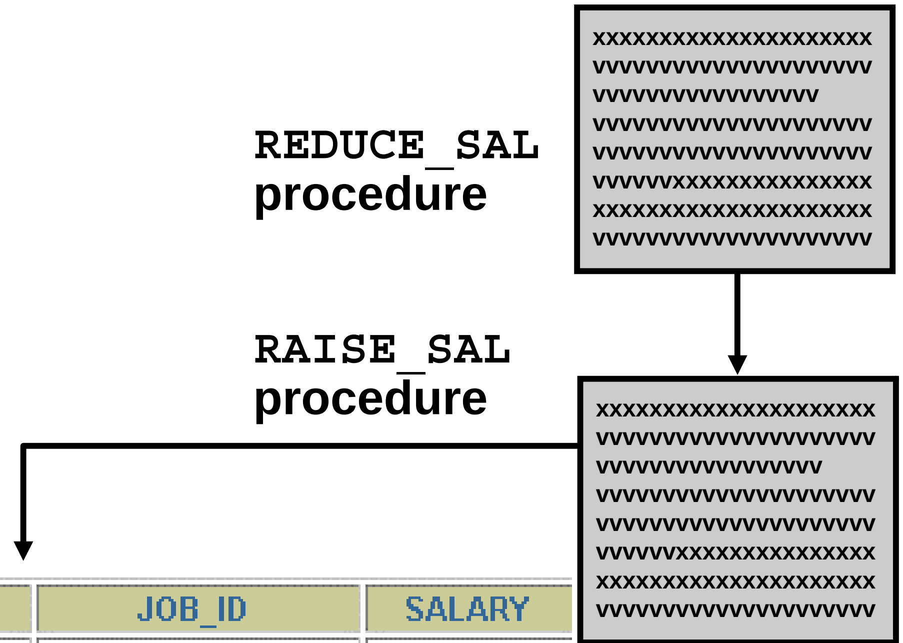
Displaying Dependencies

The 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



EMPLOYEES table

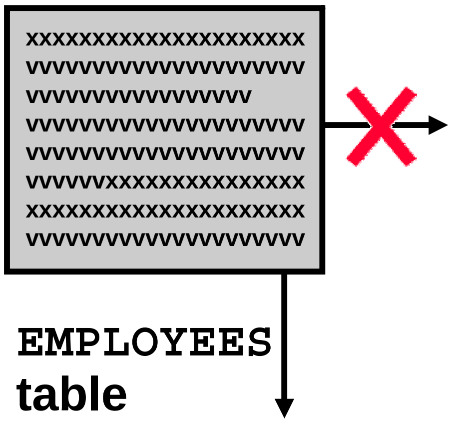
EMPLOYEE_ID	LAST_NAME	JOB_ID	SALARY
100	King	AD_PRES	24000
101	Kochhar	AD_VP	17000
102	De Haan	AD_VP	17000
103	Hunold	IT_PROG	9000
104	Ernst	IT_PROG	6000

…

ORACLE

A Scenario of Local Naming Dependencies

QUERY_EMP procedure



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_PROG	6000

...

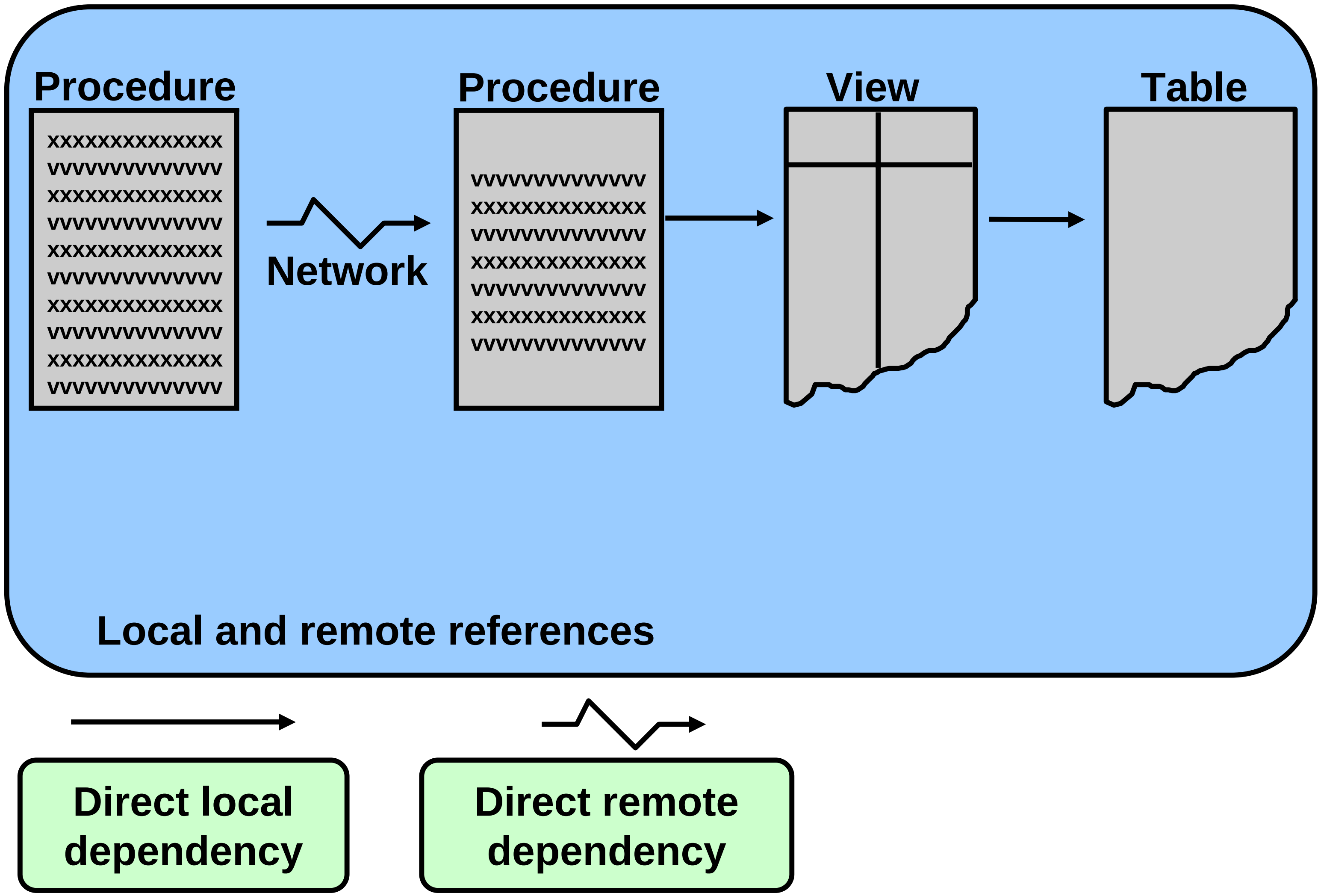
EMPLOYEES table

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_PROG	6000

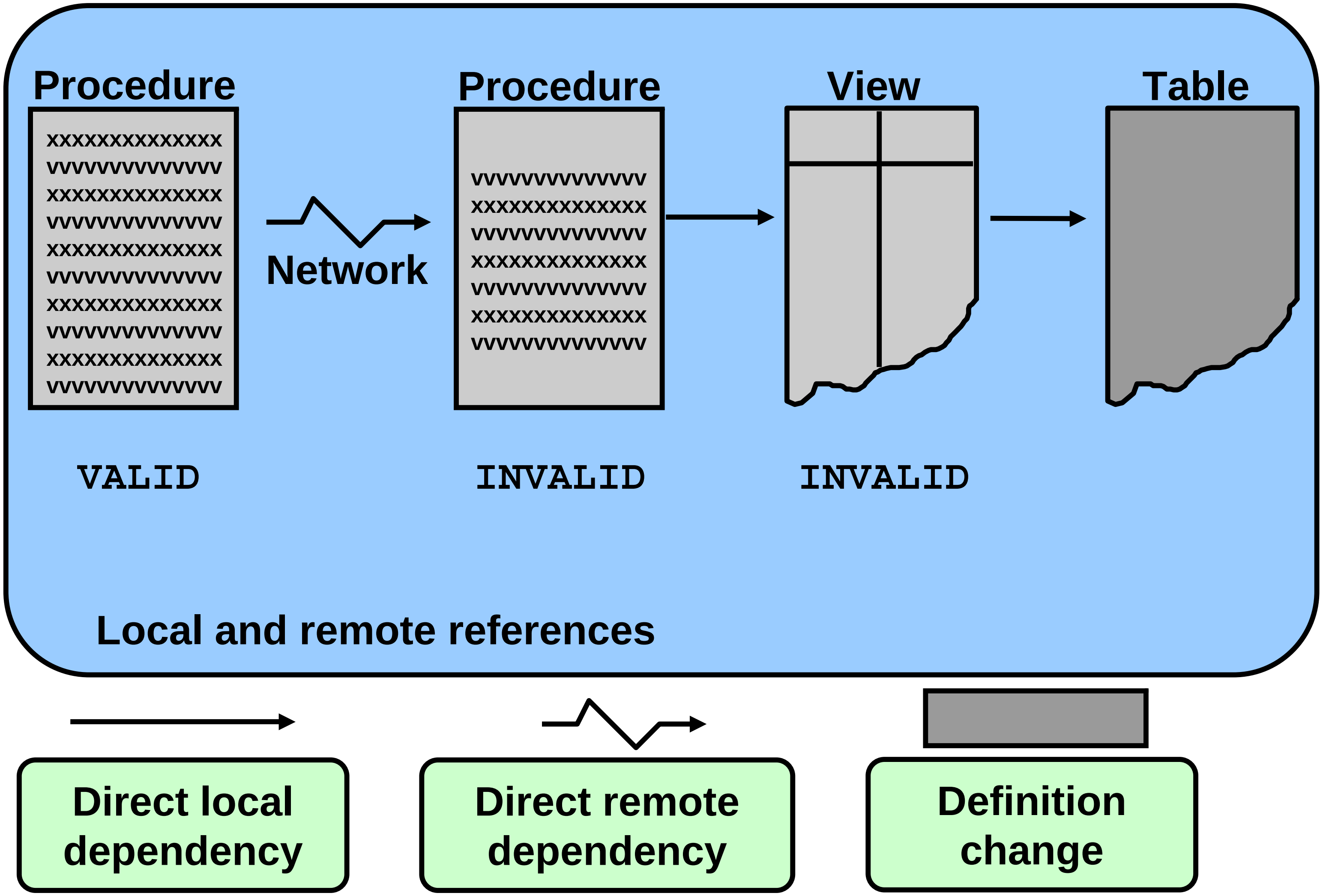
...

ORACLE

Understanding Remote Dependencies



Understanding Remote Dependencies



Concepts of Remote Dependencies

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

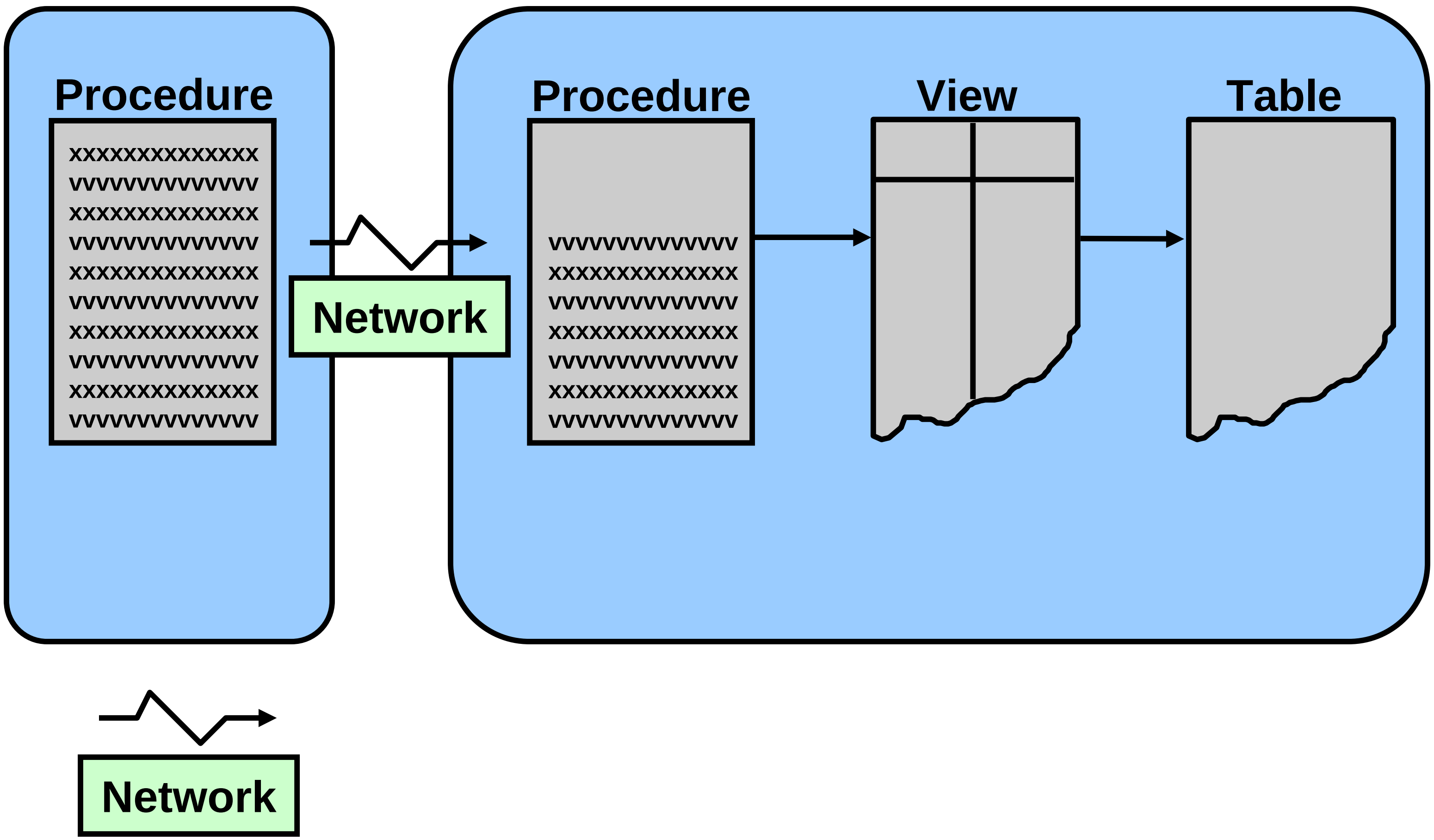
- `TIMESTAMP` **checking**
- `SIGNATURE` **checking**

`REMOTE_DEPENDENCIES_MODE` Parameter

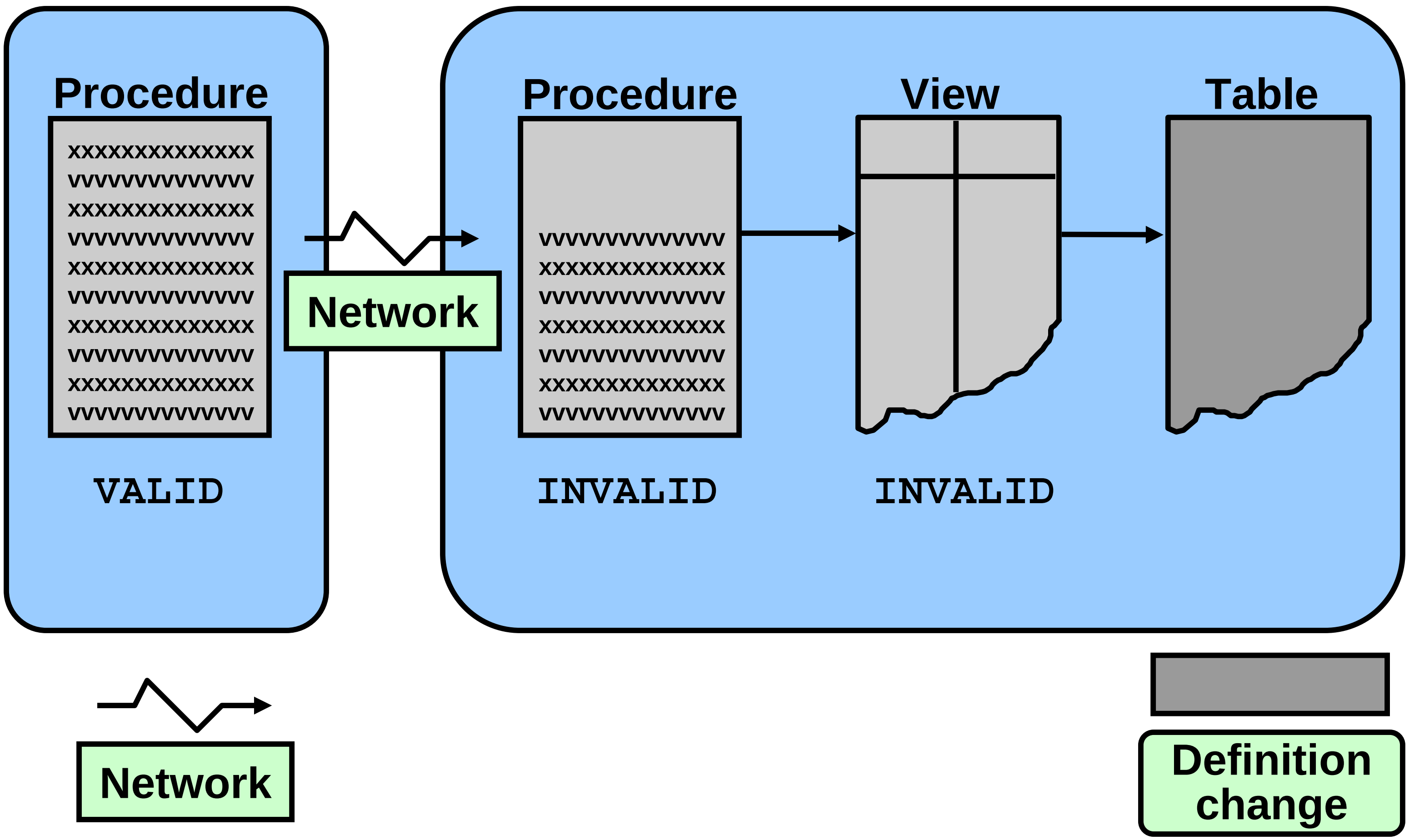
Setting `REMOTE_DEPENDENCIES_MODE`:

- **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`

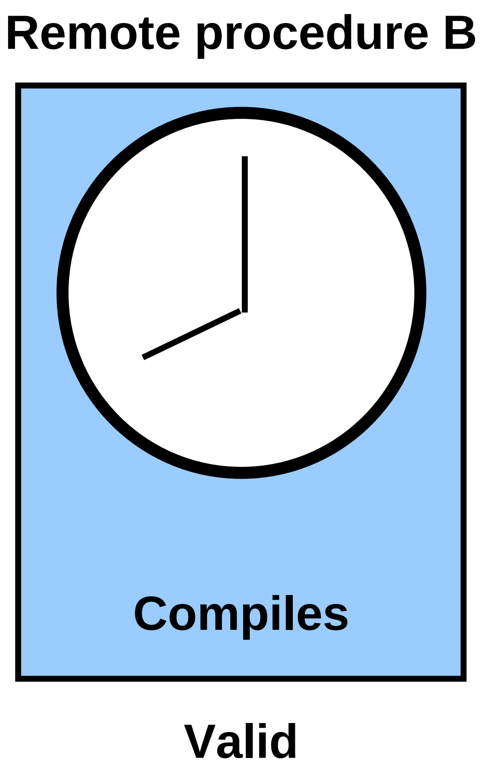
Remote Dependencies and Time Stamp Mode



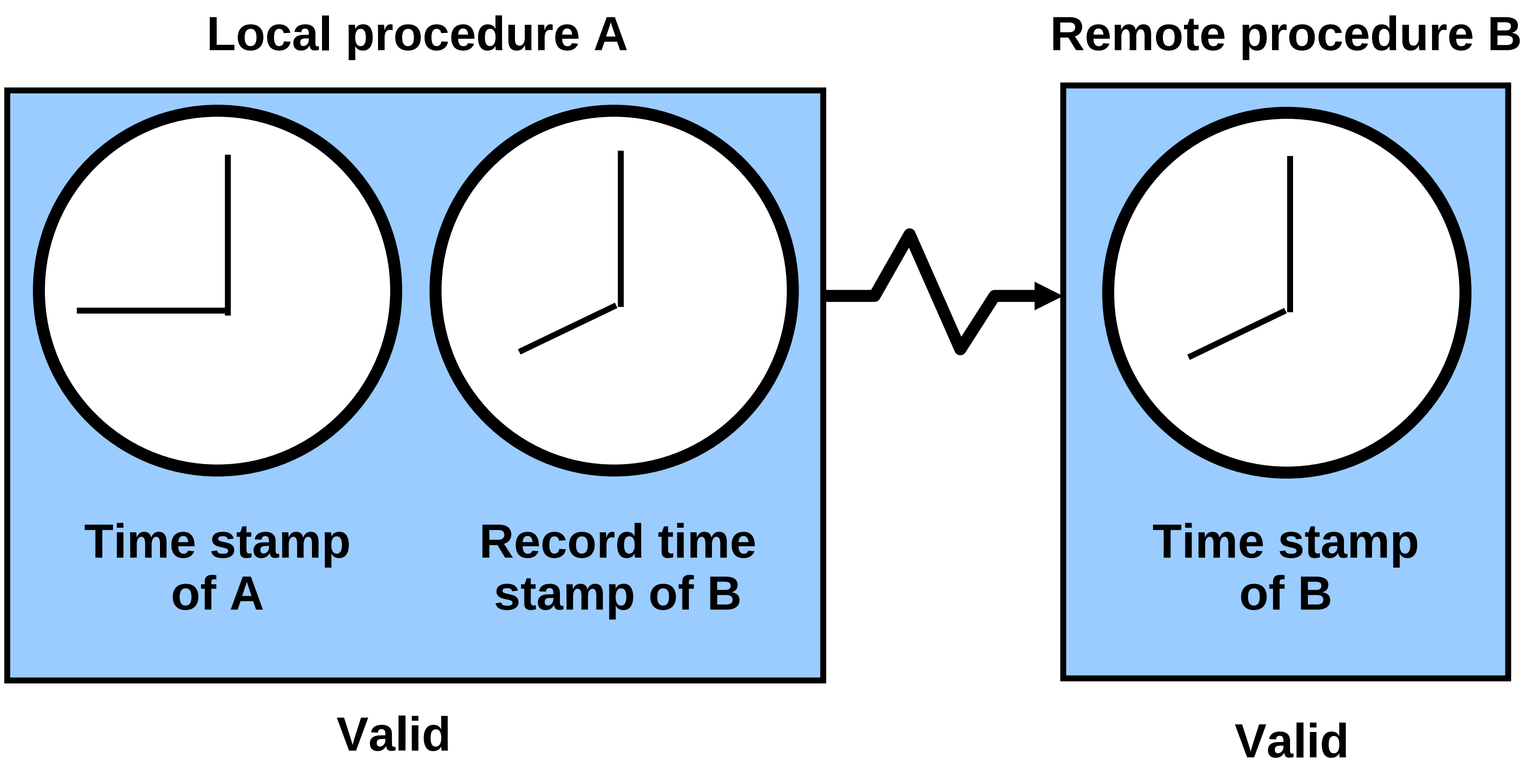
Remote Dependencies and Time Stamp Mode



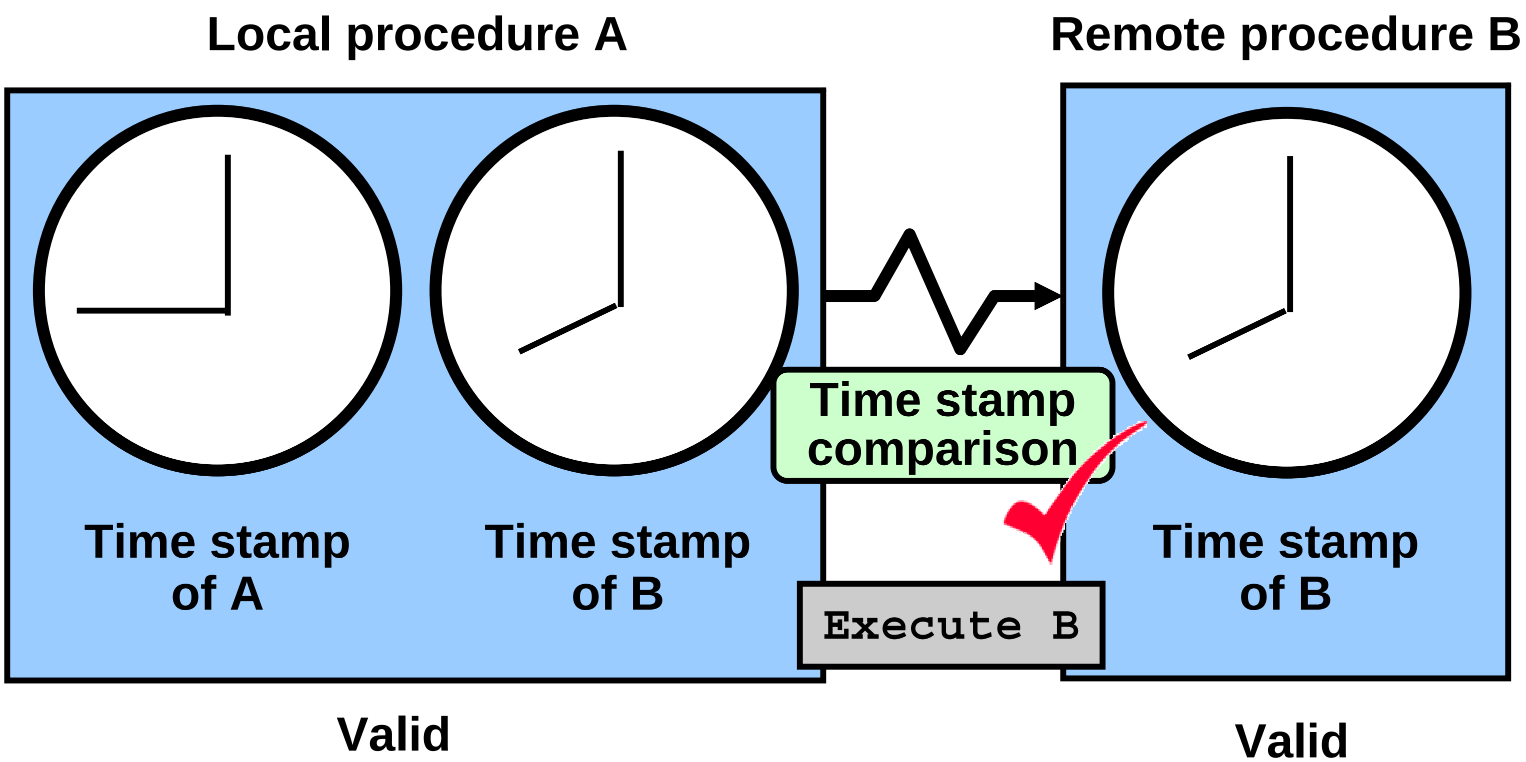
Remote Procedure B Compiles at 8:00 a.m.



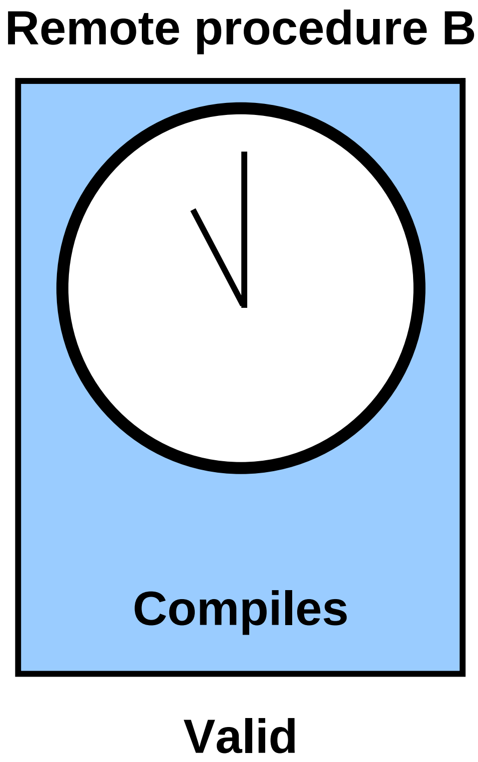
Local Procedure A
Compiles at 9:00 a.m.



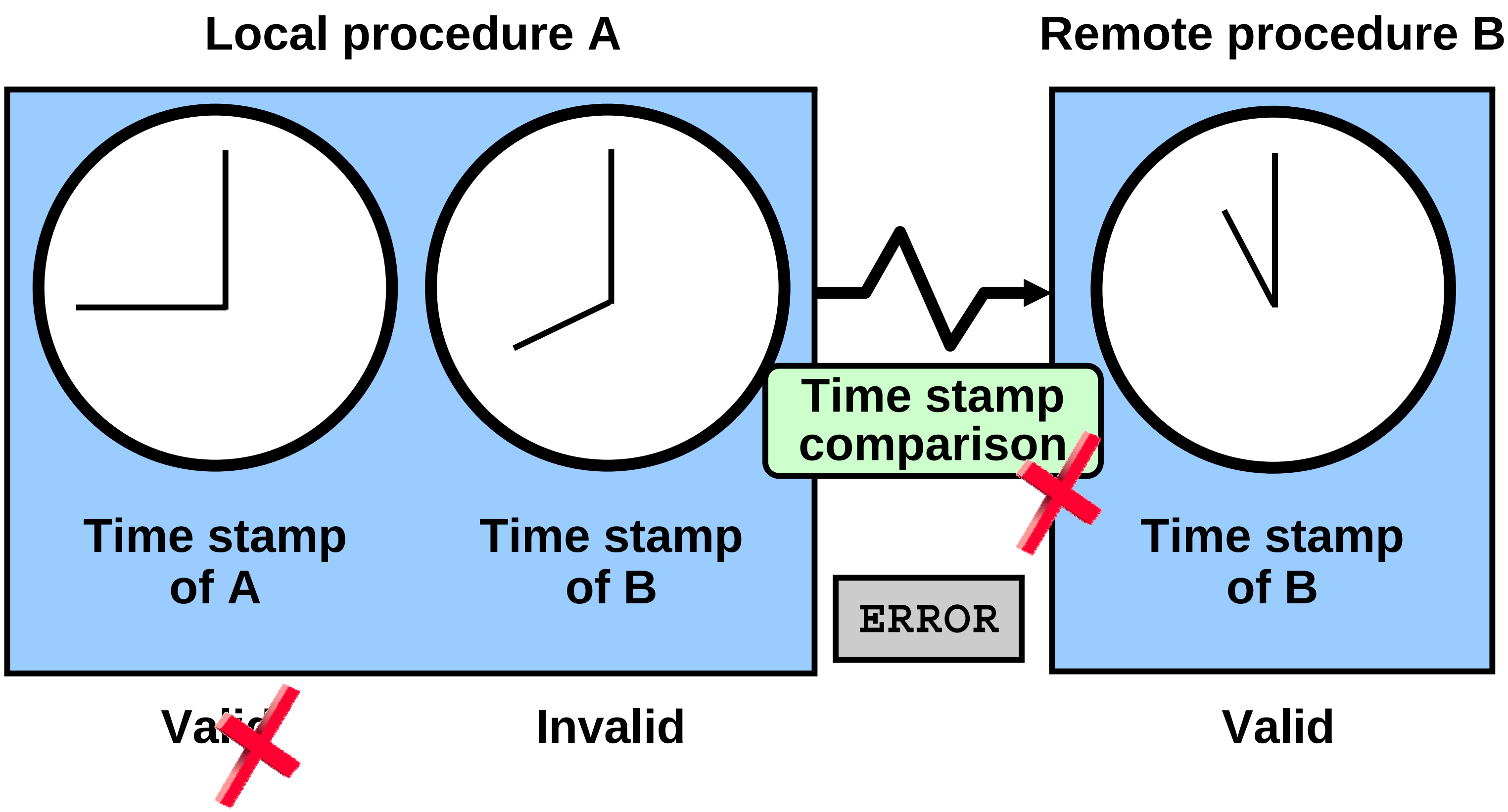
Execute Procedure A



Remote Procedure B Recompiled at 11:00 a.m.



Execute Procedure A



Signature Mode

- The signature of a procedure is:
  – The name of the procedure
  – The data types 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 | SPECIFICATION | BODY];
```

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

ORACLE

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

ORACLE

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 that has the same name and structure exists**
- **The PL/SQL body of a referenced procedure has been modified and recompiled successfully**

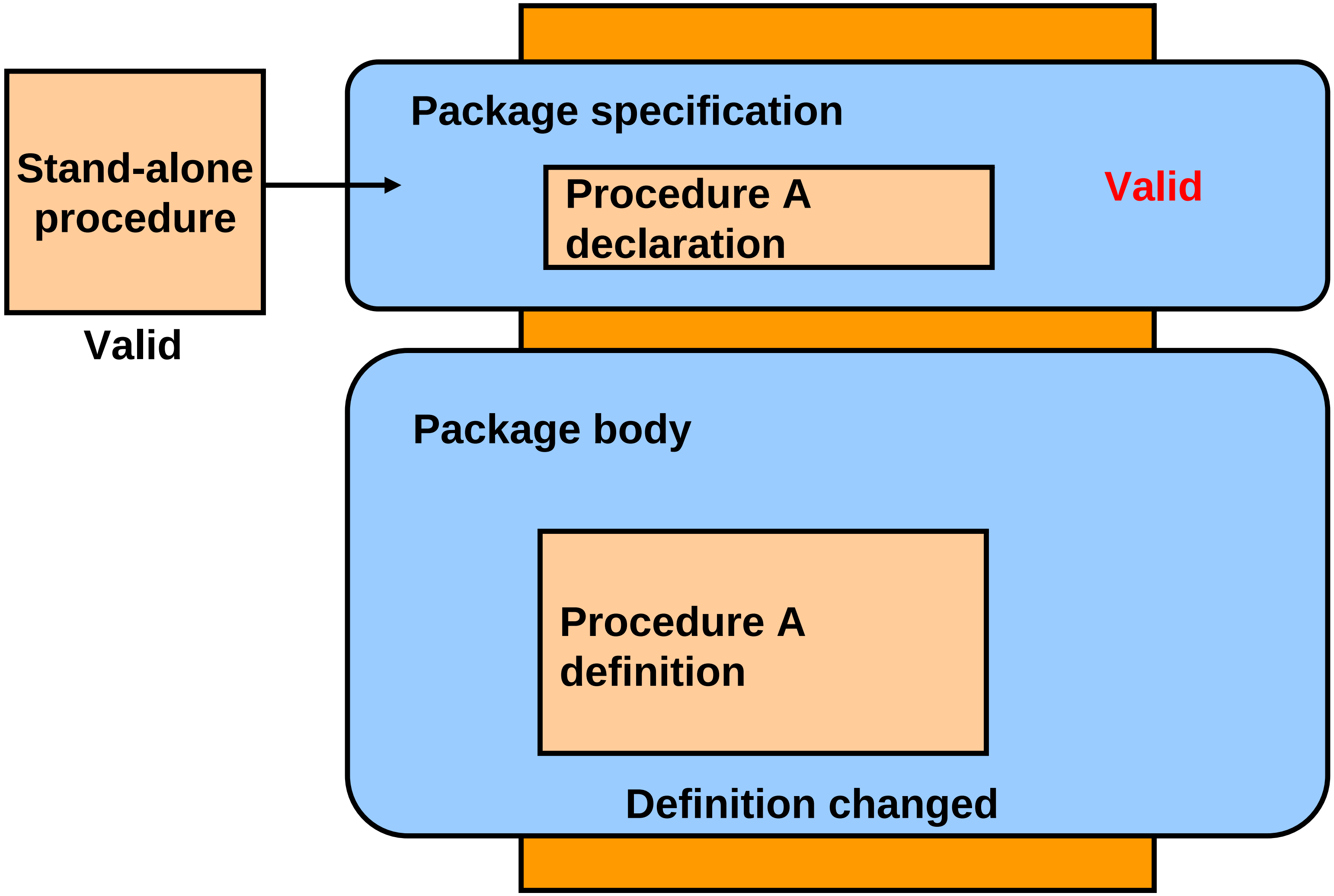
ORACLE

Recompilation of Procedures

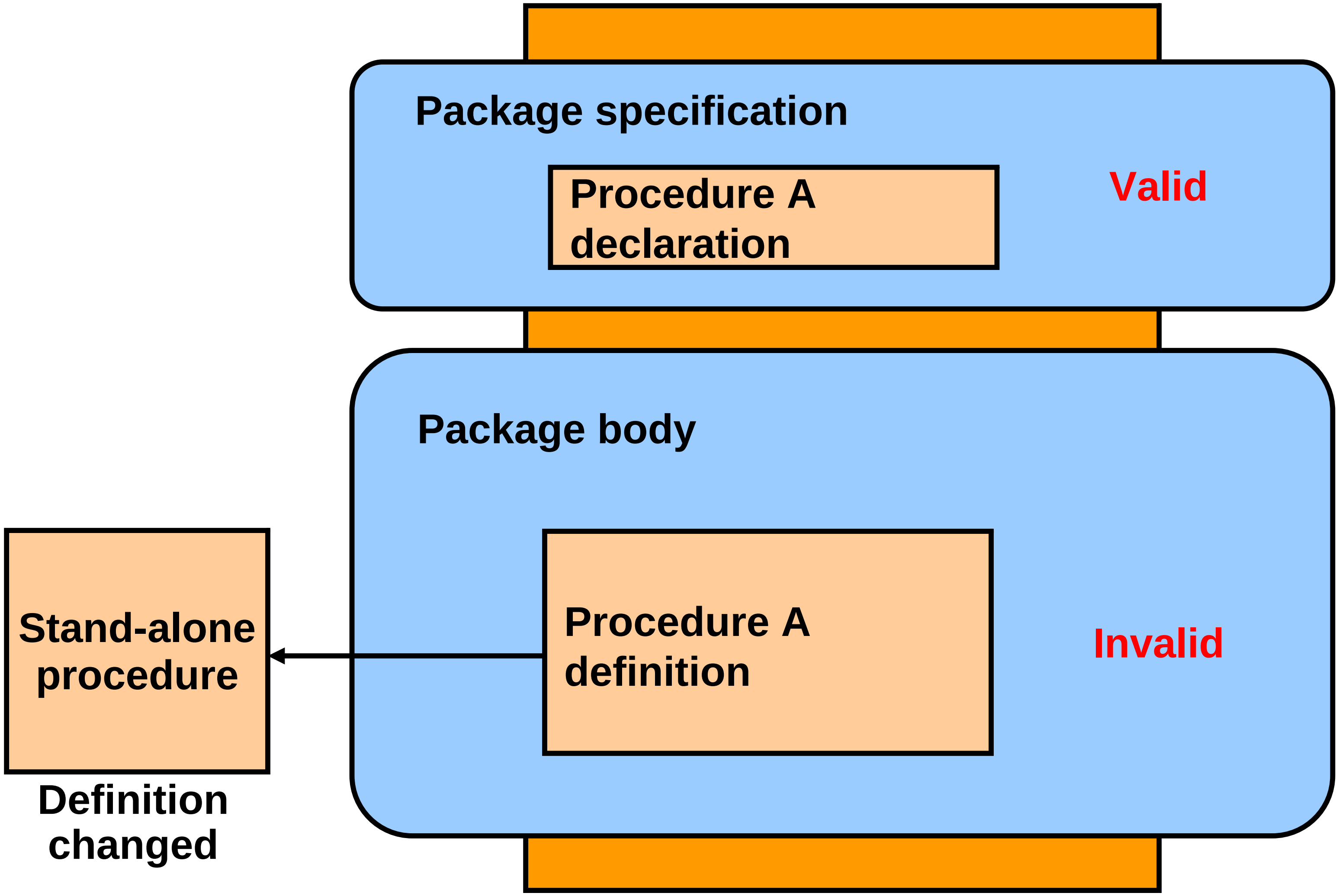
Minimize dependency failures by:

- **Declaring records with 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

ORACLE

Practice 8: Overview

This practice covers the following topics:

- **Using `DEPTREE_FILL` and `IDEPTREE` to view dependencies**
- **Recompiling procedures, functions, and packages**

9

Manipulating Large Objects

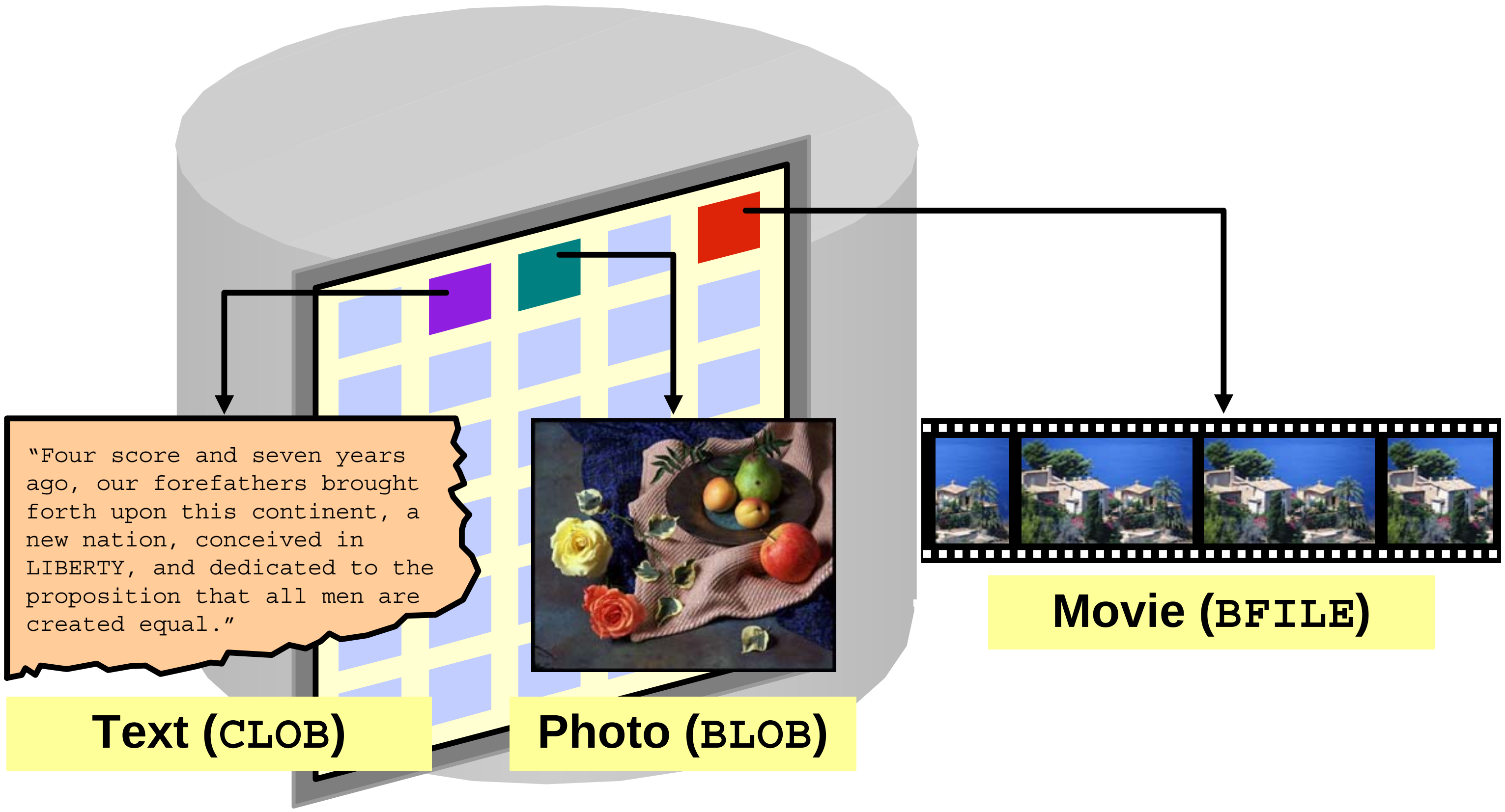
Objectives

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

- **Compare and contrast `LONG` and `LOB` (large object) data types**
- **Create and maintain `LOB` data types**
- **Differentiate between internal and external `LOB`s**
- **Use the `DBMS_LOB` PL/SQL package**
- **Describe the use of temporary `LOB`s**

What Is a LOB?

LOBs are used to store large unstructured data such as text, graphic images, films, and sound waveforms.



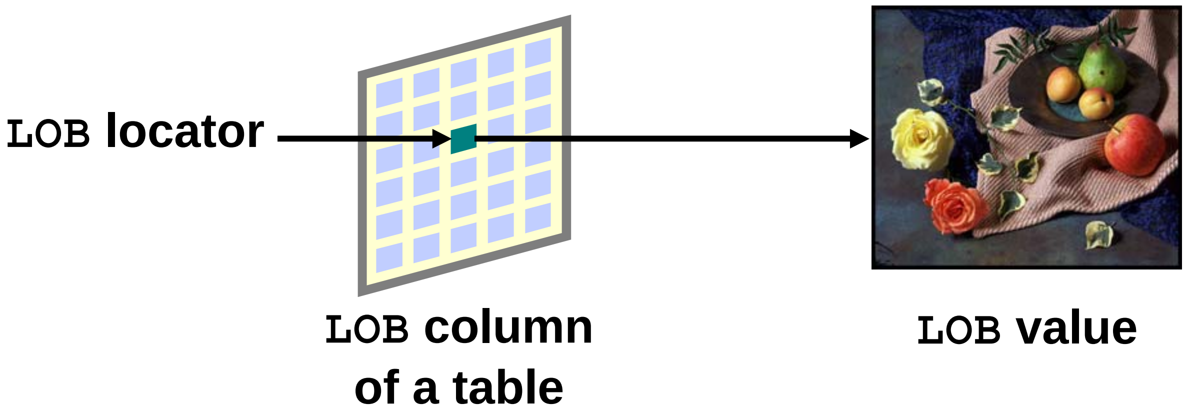
Contrasting LONG and LOB Data Types

LONG and LONG RAW	LOB
Single LONG column per table	Multiple LOB columns per table
Up to 2 GB	Up to 4 GB
SELECT returns data	SELECT returns locator
Data stored in-line	Data stored in-line or out-of-line
Sequential access to data	Random access to data

ORACLE

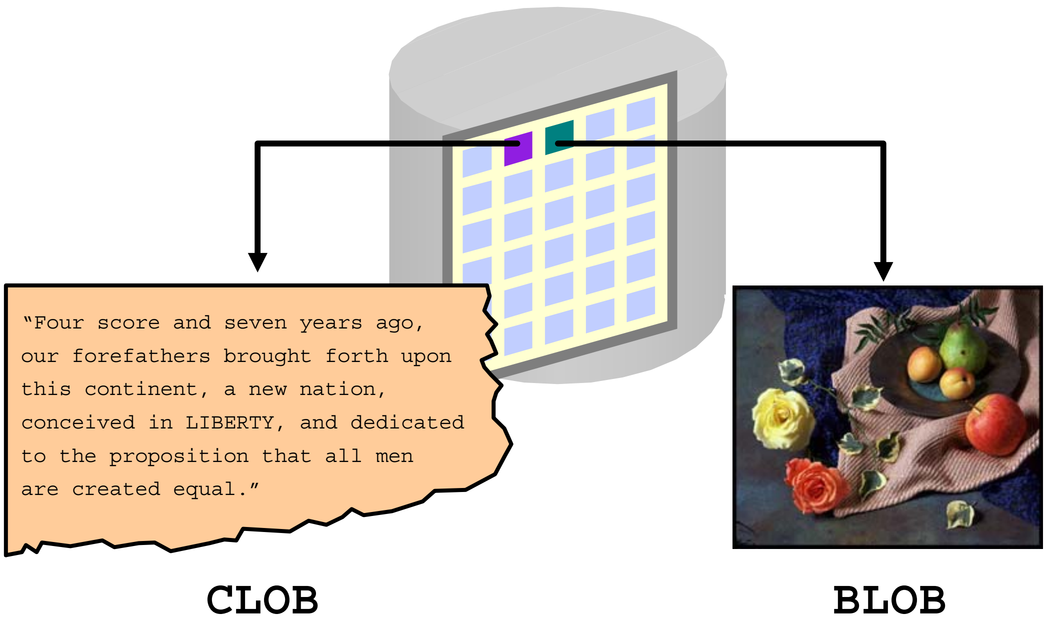
Anatomy of a LOB

The LOB column stores a locator to the LOB's value.



Internal LOBs

The LOB value is stored in the database.



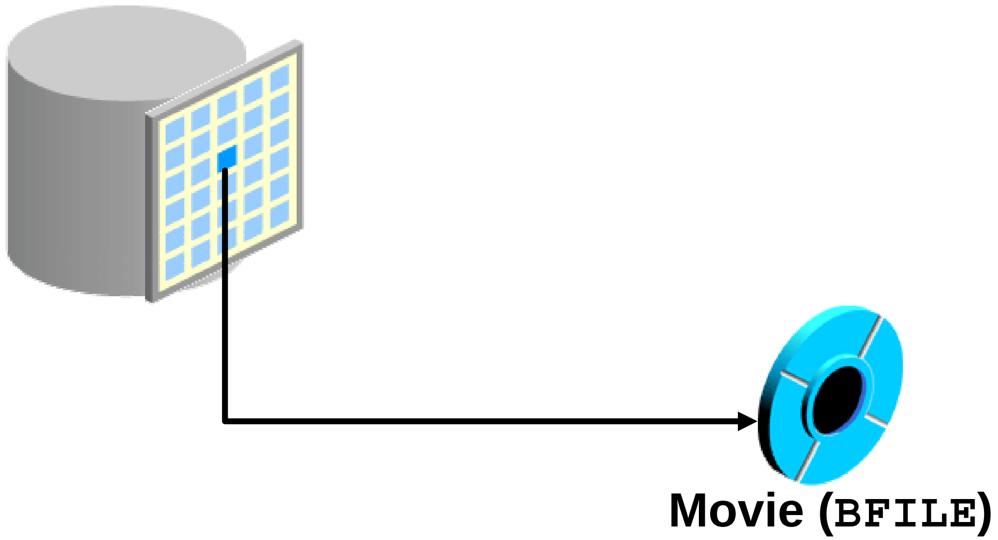
Managing Internal LOBs

- To interact fully with LOB, file-like interfaces are provided in:
  - PL/SQL package DBMS_LOB
  - Oracle Call Interface (OCI)
  - Oracle Objects for object linking and embedding (OLE)
  - Pro*C/C++ and Pro*COBOL precompilers
  - Java Database Connectivity (JDBC)
- The Oracle server provides some support for LOB management through SQL.

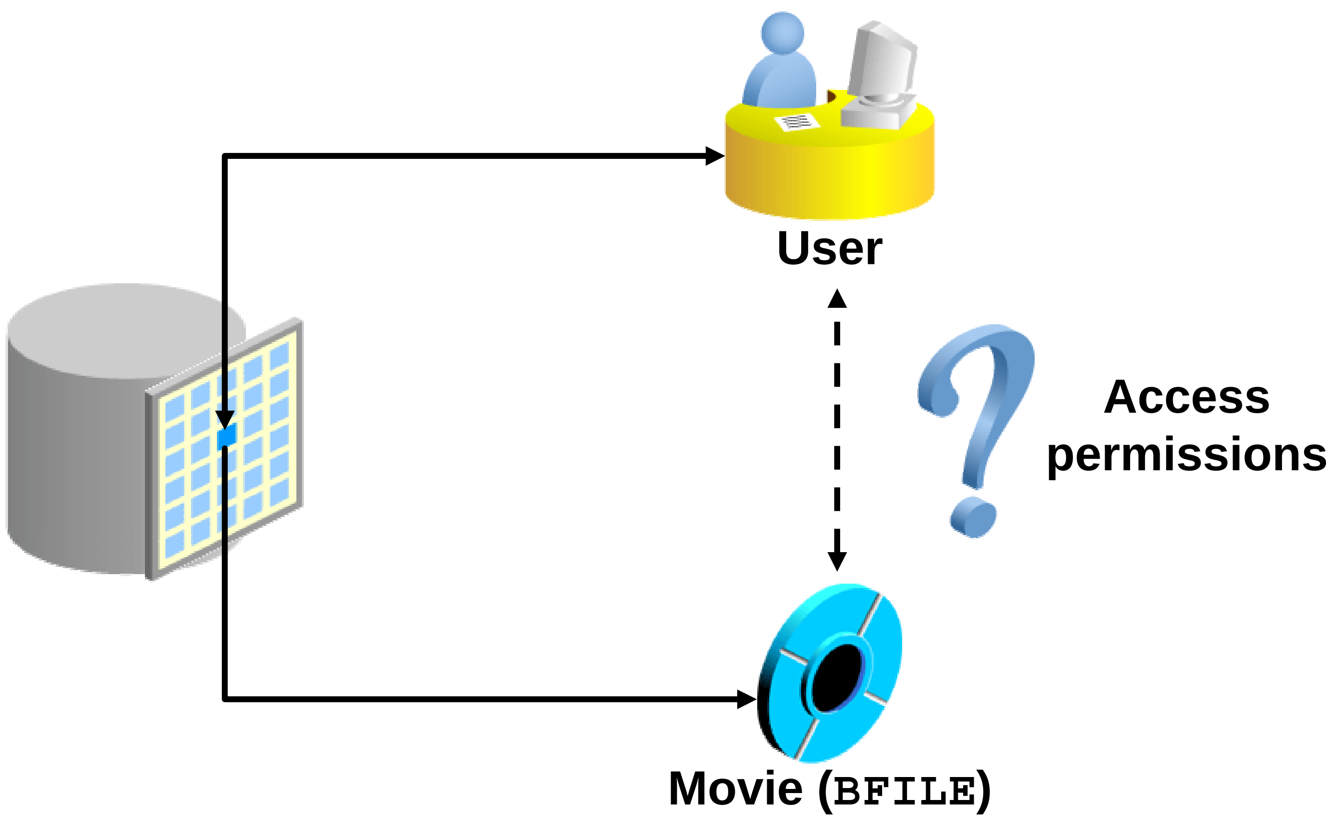
What Are BFILEs?

The `BFILE` data type supports an external or file-based large object as:

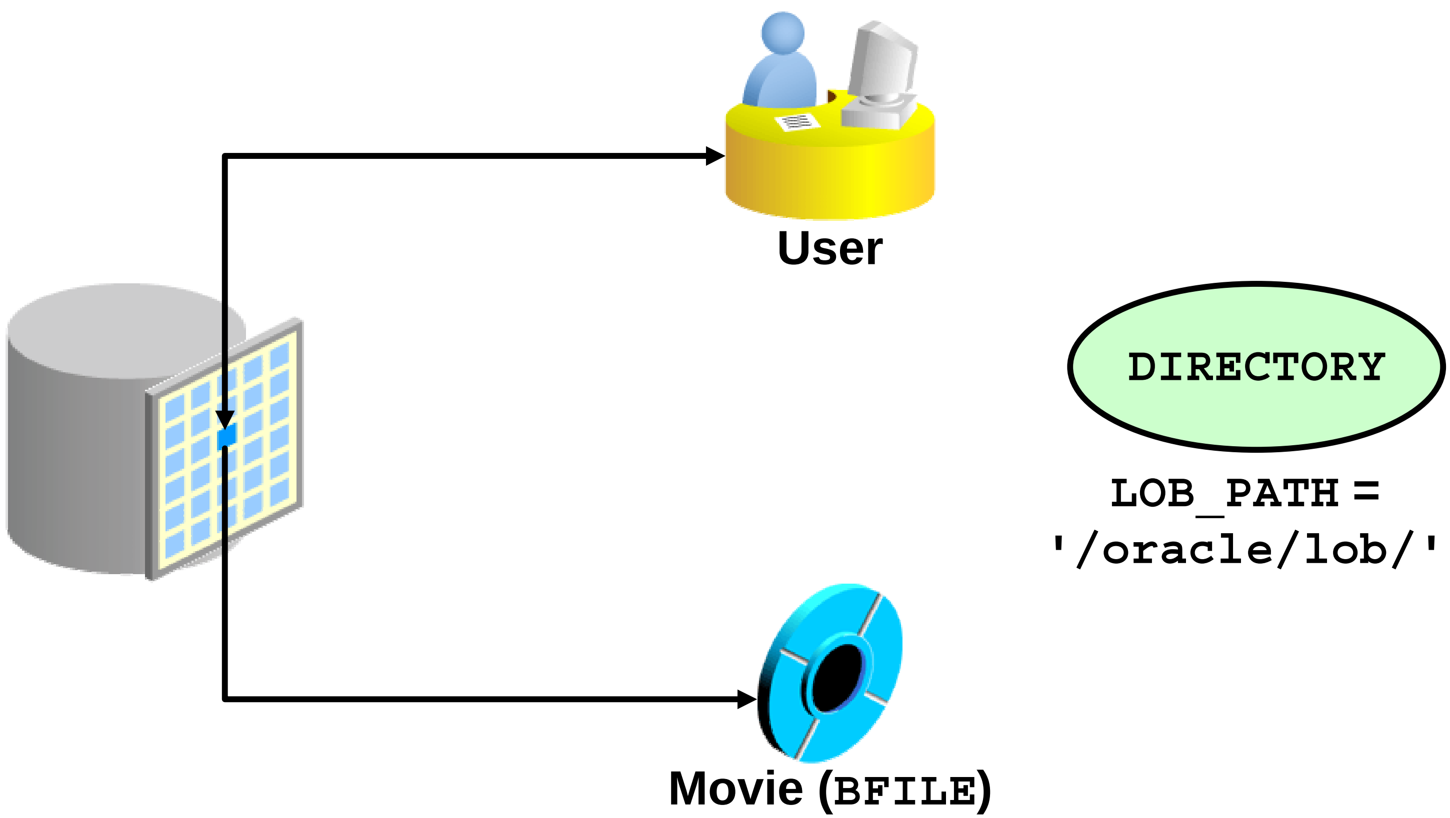
- Attributes in an object type
- Column values in a table



Securing `BFILEs`



A New Database Object: DIRECTORY



Guidelines for Creating `DIRECTORY` Objects

- **Do not create `DIRECTORY` objects on paths with database files.**
- **Limit the number of people who are given the following system privileges:**
  - **`CREATE ANY DIRECTORY`**
  - **`DROP ANY DIRECTORY`**
- **All `DIRECTORY` objects are owned by `SYS`.**
- **Create directory paths and properly set permissions before using the `DIRECTORY` object so that the Oracle server can read the file.**

Managing `BFILE`s

The DBA or the system administrator:

1. Creates an OS directory and supplies files
2. Creates a `DIRECTORY` object in the database
3. Grants the `READ` privilege on the `DIRECTORY` object to appropriate database users

The developer or the user:

4. Creates an Oracle table with a column defined as a `BFILE` data type
5. Inserts rows into the table using the `BFILENAME` function to populate the `BFILE` column
6. Writes a PL/SQL subprogram that declares and initializes a `LOB` locator, and reads `BFILE`

Preparing to Use BFILES

1. Create an OS directory to store the physical data files:

```
mkdir /temp/data_files
```

2. Create a DIRECTORY object by using the CREATE DIRECTORY command:

```
CREATE DIRECTORY data_files
              AS '/temp/data_files';
```

3. Grant the READ privilege on the DIRECTORY object to appropriate users:

```
GRANT READ ON DIRECTORY data_files
   TO SCOTT, MANAGER_ROLE, PUBLIC;
```

Populating `BFILE` Columns with SQL

- **Use the `BFILENAME` function to initialize a `BFILE` column. The function syntax is:**

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

- **Example:**
  - **Add a `BFILE` column to a table:**

```
ALTER TABLE employees ADD video BFILE;
```

  - **Update the column using the `BFILENAME` function:**

```
UPDATE employees
  SET video = BFILENAME('DATA_FILES', 'King.avi')
WHERE employee_id = 100;
```

Populating a BFILE Column with PL/SQL

```
CREATE PROCEDURE set_video(
  dir_alias VARCHAR2, dept_id NUMBER) IS
  filename VARCHAR2(40);
  file_ptr BFILE;
  CURSOR emp_csr IS
    SELECT first_name FROM employees
    WHERE department_id = dept_id FOR UPDATE;
BEGIN
  FOR rec IN emp_csr LOOP
    filename := rec.first_name || '.gif';
    file_ptr := BFILENAME(dir_alias, filename);
    DBMS_LOB.FILEOPEN(file_ptr);
    UPDATE employees SET video = file_ptr
      WHERE CURRENT OF emp_csr;
    DBMS_OUTPUT.PUT_LINE('FILE: ' || filename ||
     ' SIZE: ' || DBMS_LOB.GETLENGTH(file_ptr));
    DBMS_LOB.FILECLOSE(file_ptr);
  END LOOP;
END set_video;
```

Using DBMS_LOB Routines with BFILES

The DBMS_LOB.FILEEXISTS function can check whether the file exists in the OS. The function:

- Returns 0 if the file does not exist
- Returns 1 if the file does exist

```
CREATE FUNCTION get_filesize(file_ptr IN OUT BFILE)
RETURN NUMBER IS
  file_exists BOOLEAN;
  length NUMBER:= -1;
BEGIN
  file_exists := DBMS_LOB.FILEEXISTS(file_ptr)=1;
  IF file_exists THEN
    DBMS_LOB.FILEOPEN(file_ptr);
    length := DBMS_LOB.GETLENGTH(file_ptr);
    DBMS_LOB.FILECLOSE(file_ptr);
  END IF;
  RETURN length;
END;
/
```

ORACLE

Migrating from `LONG` to `LOB`

Oracle Database 10*g* enables the migration of `LONG` columns to `LOB` columns.

- **Data migration consists of the procedure to move existing tables containing `LONG` columns to use `LOB`s:**

```
ALTER TABLE [<schema>.] <table_name>
    MODIFY (<long_col_name> {CLOB | BLOB | NCLOB})
```

- **Application migration consists of changing existing `LONG` applications for using `LOB`s.**

Migrating from LONG to LOB

- **Implicit conversion: From LONG (LONG RAW) or a VARCHAR2 (RAW) variable to a CLOB (BLOB) variable, and vice versa**
- **Explicit conversion:**
  - **TO_CLOB() converts LONG, VARCHAR2, and CHAR to CLOB.**
  - **TO_BLOB() converts LONG RAW and RAW to BLOB.**
- **Function and procedure parameter passing:**
  - **CLOBs and BLOBs are passed as actual parameters**
  - **VARCHAR2, LONG, RAW, and LONG RAW are formal parameters, and vice versa.**
- **LOB data is acceptable in most of the SQL and PL/SQL operators and built-in functions.**

DBMS_LOB Package

- **Working with LOBs often requires the use of the Oracle-supplied DBMS_LOB package.**
- **DBMS_LOB provides routines to access and manipulate internal and external LOBs.**
- **Oracle Database 10*g* enables retrieving LOB data directly using SQL without a special LOB API.**
- **In PL/SQL, you can define a VARCHAR2 for a CLOB and a RAW for a BLOB.**

DBMS_LOB Package

- **Modify LOB values:**
  APPEND, COPY, ERASE, TRIM, WRITE, LOADFROMFILE
- **Read or examine LOB values:**
  GETLENGTH, INSTR, READ, SUBSTR
- **Specific to BFILEs:**
  FILECLOSE, FILECLOSEALL, FILEEXISTS, FILEGETNAME, FILEISOPEN, FILEOPEN

DBMS_LOB Package

- NULL parameters get NULL returns.
- Offsets:
  - BLOB, BFILE: Measured in bytes
  - CLOB, NCLOB: Measured in characters
- There are no negative values for parameters.

DBMS_LOB.READ and DBMS_LOB.WRITE

```
PROCEDURE READ (
  lobsrc IN BFILE|BLOB|CLOB ,
  amount IN OUT BINARY_INTEGER,
  offset IN INTEGER,
  buffer OUT RAW|VARCHAR2 )
```

```
PROCEDURE WRITE (
  lobdst IN OUT BLOB|CLOB,
  amount IN OUT BINARY_INTEGER,
  offset IN INTEGER := 1,
  buffer IN RAW|VARCHAR2 )   -- RAW for BLOB
```

Initializing LOB Columns Added to a Table

- Create the table with columns using the LOB type, or add the LOB columns using ALTER TABLE.

```
ALTER TABLE employees
     ADD (resume CLOB, picture BLOB);
```

- Initialize the column LOB locator value with the DEFAULT option or DML statements using:
  - EMPTY_CLOB() function for a CLOB column
  - EMPTY_BLOB() function for a BLOB column

```
CREATE TABLE emp_hiredata (
   employee_id  NUMBER(6),
   full_name    VARCHAR2(45),
   resume       CLOB DEFAULT EMPTY_CLOB(),
   picture      BLOB DEFAULT EMPTY_BLOB());
```

ORACLE

Populating `LOB` Columns

- **Insert a row into a table with `LOB` columns:**

```
INSERT INTO emp_hiredata
  (employee_id, full_name, resume, picture)
 VALUES (405, 'Marvin Ellis', EMPTY_CLOB(), NULL);
```

- **Initialize a `LOB` using the `EMPTY_BLOB()` function:**

```
UPDATE emp_hiredata
 SET resume = 'Date of Birth: 8 February 1951',
    picture = EMPTY_BLOB()
 WHERE employee_id = 405;
```

- **Update a `CLOB` column:**

```
UPDATE emp_hiredata
 SET resume = 'Date of Birth: 1 June 1956'
 WHERE employee_id = 170;
```

Updating LOB by Using DBMS_LOB in PL/SQL

```
DECLARE
  lobloc CLOB;        -- serves as the LOB locator
  text   VARCHAR2(50) := 'Resigned = 5 June 2000';
  amount NUMBER ;     -- amount to be written
  offset INTEGER;     -- where to start writing
BEGIN
  SELECT resume INTO lobloc FROM emp_hiredata
  WHERE employee_id = 405 FOR UPDATE;
  offset := DBMS_LOB.GETLENGTH(lobloc) + 2;
  amount := length(text);
  DBMS_LOB.WRITE (lobloc, amount, offset, text);
  text := ' Resigned = 30 September 2000';
  SELECT resume INTO lobloc FROM emp_hiredata
  WHERE employee_id = 170 FOR UPDATE;
  amount := length(text);
  DBMS_LOB.WRITEAPPEND(lobloc, amount, text);
  COMMIT;
END;
```

Selecting `CLOB` Values by Using SQL

```
SELECT employee_id, full_name , resume -- CLOB
FROM emp_hiredata
WHERE employee_id IN (405, 170);
```

EMPLOYEE_ID	FULL_NAME	RESUME
405	Marvin Ellis	Date of Birth: 8 February 1951 Resigned = 5 June 2000
170	Joe Fox	Date of Birth: 1 June 1956 Resigned = 30 September 2000

Selecting `CLOB` Values by Using `DBMS_LOB`

- `DBMS_LOB.SUBSTR` (lob, amount, start_pos)
- `DBMS_LOB.INSTR` (lob, pattern)

```
SELECT DBMS_LOB.SUBSTR (resume, 5, 18),
       DBMS_LOB.INSTR (resume,' = ')
FROM   emp_hiredata
WHERE  employee_id IN (170, 405);
```

DBMS_LOB.SUBSTR(RESUME,5,18)	DBMS_LOB.INSTR(RESUME,'=')
Febru	40
June	36

Selecting CLOB Values in PL/SQL

```
SET LINESIZE 50 SERVEROUTPUT ON FORMAT WORD_WRAP
DECLARE
  text VARCHAR2(4001);
BEGIN
 SELECT resume INTO text
 FROM emp_hiredata
 WHERE employee_id = 170;
 DBMS_OUTPUT.PUT_LINE('text is: '|| text);
END;
/
```

```
text is: Date of Birth: 1 June 1956 Resigned = 30
September 2000

PL/SQL procedure successfully completed.
```

Removing `LOBS`

- **Delete a row containing `LOBS`:**

```
DELETE
FROM  emp_hiredata
WHERE employee_id = 405;
```

- **Disassociate a `LOB` value from a row:**

```
UPDATE emp_hiredata
SET resume = EMPTY_CLOB()
WHERE employee_id = 170;
```

Temporary LOBs

- Temporary LOBs:
  - Provide an interface to support creation of LOBs that act like local variables
  - Can be BLOBs, CLOBs, or NCLOBs
  - Are not associated with a specific table
  - Are created using the DBMS_LOB.CREATETEMPORARY procedure
  - Use DBMS_LOB routines
- The lifetime of a temporary LOB is a session.
- Temporary LOBs are useful for transforming data in permanent internal LOBs.

Creating a Temporary LOB

PL/SQL procedure to create and test a temporary LOB:

```
CREATE OR REPLACE PROCEDURE is_templob_open(
  lob IN OUT BLOB, retval OUT INTEGER) IS
BEGIN
  -- create a temporary LOB
  DBMS_LOB.CREATETEMPORARY (lob, TRUE);
  -- see if the LOB is open: returns 1 if open
  retval := DBMS_LOB.ISOPEN (lob);
  DBMS_OUTPUT.PUT_LINE (
    'The file returned a value...' || retval);
  -- free the temporary LOB
  DBMS_LOB.FREETEMPORARY (lob);
END;
/
```

Summary

In this lesson, you should have learned how to:

- Identify four built-in types for large objects: `BLOB`, `CLOB`, `NCLOB`, and `BFILE`
- Describe how `LOB`s replace `LONG` and `LONG RAW`
- Describe two storage options for `LOB`s:
  - Oracle server (internal `LOB`s)
  - External host files (external `LOB`s)
- Use the `DBMS_LOB` PL/SQL package to provide routines for `LOB` management
- Use temporary `LOB`s in a session

Practice 9: Overview

This practice covers the following topics:

- Creating object types using the `CLOB` and `BLOB` data types
- Creating a table with `LOB` data types as columns
- Using the `DBMS_LOB` package to populate and interact with the `LOB` data

10

Creating Triggers

Objectives

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

- Describe the different types of triggers
- Describe database triggers and their uses
- 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 database
- Executes implicitly whenever a particular event takes place
- Can be either of the following:
  – 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

- You can design triggers to:
  - Perform related actions
  - Centralize global operations
- You must not design triggers:
  - Where functionality is already built into the Oracle server
  - That duplicate other triggers
- You can 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.

ORACLE

Creating DML Triggers

Create DML statement or row type triggers by using:

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

- A statement trigger fires once for a DML statement.
- A row trigger fires once for each row affected.

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

Types of DML Triggers

The trigger type determines whether the body executes for each row or only once for the triggering statement.

- A statement trigger:
  - Executes once for the triggering event
  - Is the default type of trigger
  - Fires once even if no rows are affected at all
- A row trigger:
  - Executes once for each row affected by the triggering event
  - Is not executed if the triggering event does not affect any rows
  - Is indicated by specifying the `FOR EACH ROW` clause

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.**

Note: If multiple triggers are defined for the same object, then the order of firing triggers is arbitrary.

Trigger-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);
```

Triggering action

BEFORE statement trigger

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

BEFORE row trigger

AFTER row trigger

AFTER statement trigger

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;
```

→ **BEFORE statement trigger**

EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID
114	Raphaely	30
115	Khoo	30
116	Baida	30
117	Tobias	30
118	Himuro	30
119	Colmenares	30

→ **BEFORE row trigger**

→ **AFTER row trigger**

...

→ **BEFORE row trigger**

→ **AFTER row trigger**

...

→ **AFTER statement trigger**

Trigger Event Types and Body

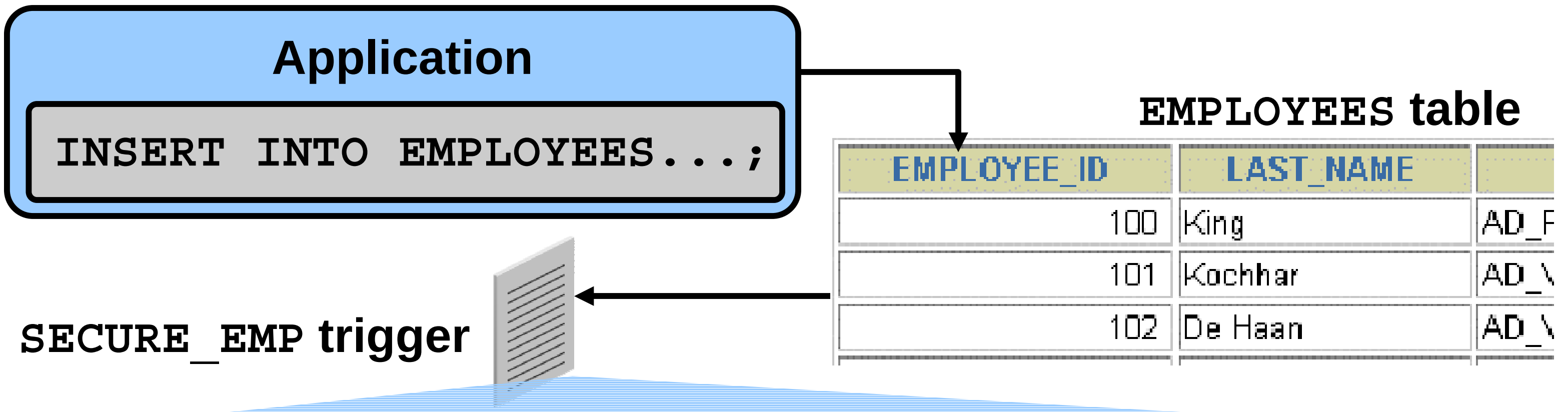
A trigger event:

- **Determines which DML statement causes the trigger to execute**
- **Types are:**
  - `INSERT`
  - `UPDATE [OF column]`
  - `DELETE`

A trigger body:

- **Determines what action is performed**
- **Is a PL/SQL block or a `CALL` to a procedure**

Creating a DML Statement Trigger




```
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;
```

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'
```

Using Conditional Predicates

```
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

```
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 more than $15,000.');
  END IF;
END;
/
```

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(user_name, time_stamp, 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;
/
```

Using `OLD` and `NEW` Qualifiers: Example Using `AUDIT_EMP`

```
INSERT INTO employees
 (employee_id, last_name, job_id, salary, ...)
VALUES (999, 'Temp emp', 'SA_REP', 6000,...);

UPDATE employees
 SET salary = 7000, last_name = 'Smith'
 WHERE employee_id = 999;
```

```
SELECT user_name, timestamp, ...
FROM audit_emp;
```

USER_NAME	TIME_STAMP	ID	OLD_LAST_NAME	NEW_LAST_NAME	OLD_TITLE	NEW_TITLE	OLD_SALARY	NEW_SALARY
ORA25	31-MAR-06			Temp emp		SA_REP		6000
ORA25	31-MAR-06	999	Temp emp	Smith	SA_REP	SA_REP	6000	7000

Restricting a Row Trigger: Example

```
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;
/
```

Summary of the Trigger Execution Model

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

Note: Integrity checking can be deferred until the `COMMIT` operation is performed.

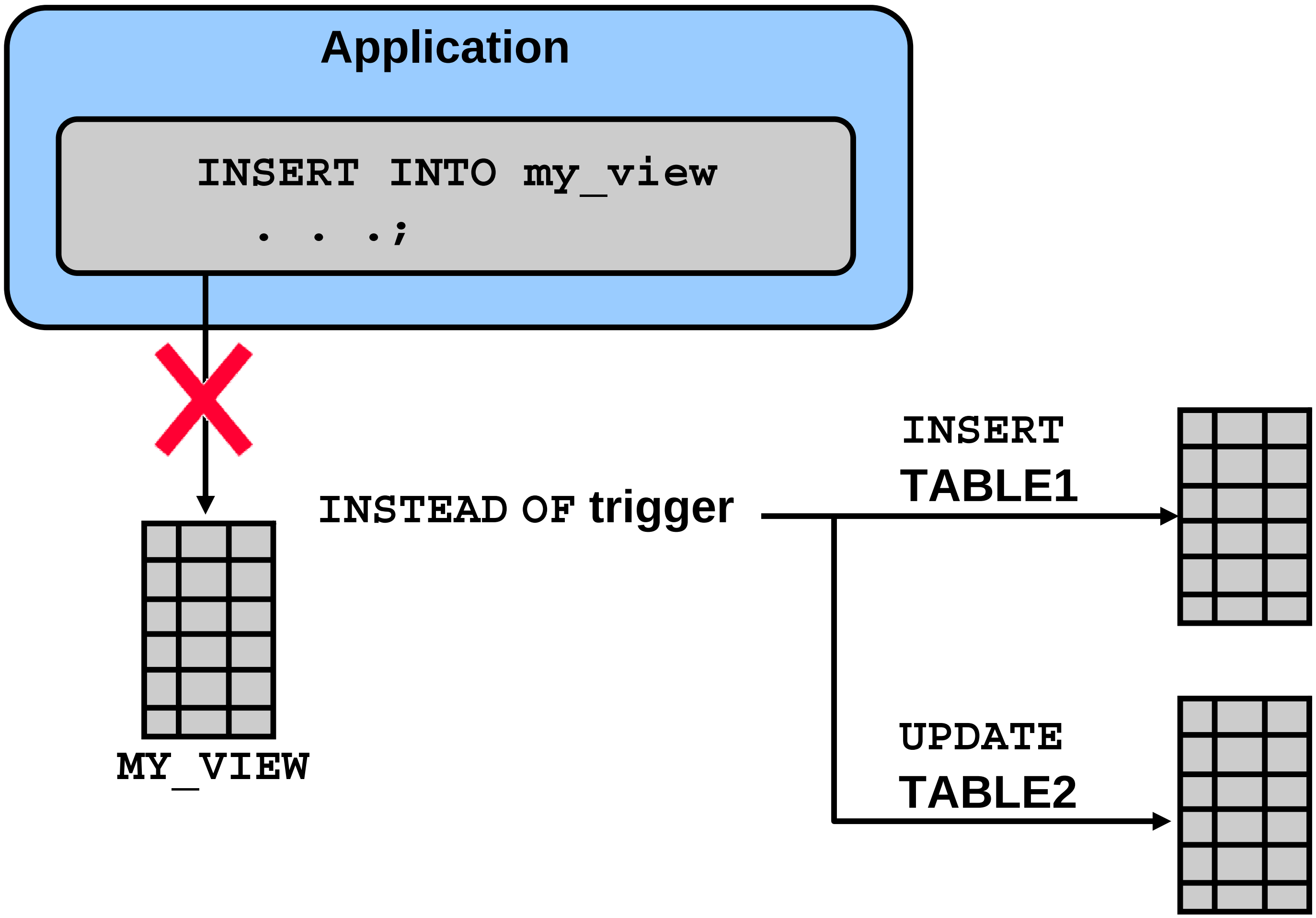
Implementing an Integrity Constraint with a Trigger

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

```
CREATE OR REPLACE TRIGGER employee_dept_fk_trg
AFTER UPDATE OF department_id
ON employees FOR EACH ROW
BEGIN
 INSERT INTO departments VALUES(:new.department_id,
          'Dept '||:new.department_id, NULL, NULL);
EXCEPTION
   WHEN DUP_VAL_ON_INDEX THEN
    NULL; -- mask exception if department exists
END;
/
```

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

INSTEAD OF Triggers



Creating an INSTEAD OF Trigger

Perform the INSERT into EMP_DETAILS that is based on EMPLOYEES and DEPARTMENTS tables:

```
INSERT INTO emp_details
VALUES (9001,'ABBOTT',3000, 10, 'Administration');
```

1 INSTEAD OF INSERT into EMP_DETAILS

EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID
100	King	90
101	Kochhar	90
102	De Haan	90

2 INSERT into NEW_EMPS

EMPLOYEE_ID	LAST_NAME	SALARY	DEPARTMENT_ID
100	King	24000	90
101	Kochhar	17000	90
102	De Haan	17000	90
...			
9001	ABBOTT	3000	10

3 UPDATE NEW_DEPTS

DEPARTMENT_ID	DEPARTMENT_NAME	DEPT_SA
10	Administration	940
20	Marketing	1900
30	Purchasing	3012
40	Human Resources	650
...		

Creating an INSTEAD OF Trigger

Use INSTEAD OF to perform DML on complex views:

```
CREATE TABLE new_emps AS
 SELECT employee_id,last_name,salary,department_id
 FROM employees;

CREATE TABLE new_depts AS
 SELECT d.department_id,d.department_name,
        sum(e.salary) dept_sal
 FROM employees e, departments d
 WHERE e.department_id = d.department_id;

CREATE VIEW emp_details AS
 SELECT e.employee_id, e.last_name, e.salary,
        e.department_id, d.department_name
 FROM employees e, departments d
 WHERE e.department_id = d.department_id
GROUP BY d.department_id,d.department_name;
```

Comparison of Database Triggers and Stored Procedures

Triggers	Procedures
Defined with `CREATE TRIGGER`	Defined with `CREATE PROCEDURE`
Data dictionary contains source code in `USER_TRIGGERS`.	Data dictionary contains source code in `USER_SOURCE`.
Implicitly invoked by DML	Explicitly invoked
`COMMIT`, `SAVEPOINT`, and `ROLLBACK` are not allowed.	`COMMIT`, `SAVEPOINT`, and `ROLLBACK` are allowed.

Comparison of Database Triggers and Oracle Forms Triggers

```
INSERT INTO EMPLOYEES
  . . .;
```

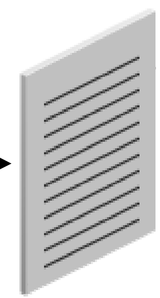
`CHECK_SAL` trigger

`EMPLOYEES` table

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_PROG	6000

…

`BEFORE INSERT` row



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
```

Removing Triggers

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

```
DROP TRIGGER trigger_name;
```

Example:

```
DROP TRIGGER secure_emp;
```

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

Testing Triggers

- 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 on other triggers.
- Test the effect of other triggers on the trigger.

Summary

In this lesson, you should have learned how to:

- Create database triggers that are invoked by DML operations
- Create statement and row trigger types
- Use database trigger-firing rules
- Enable, disable, and manage database triggers
- Develop a strategy for testing triggers
- Remove database triggers

Practice 10: Overview

This practice covers the following topics:

- **Creating row triggers**
- **Creating a statement trigger**
- **Calling procedures from a trigger**

11

Applications for Triggers

ORACLE

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 a 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

Syntax:

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

LOGON and LOGOFF Triggers: 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
/
```

Note: There is no semicolon at the end of the `CALL` statement.

ORACLE

Reading Data from a Mutating Table

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

EMPLOYEES table

Failure

CHECK_SALARY trigger

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
...			

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
  minsalary employees.salary%TYPE;
  maxsalary employees.salary%TYPE;
BEGIN
  SELECT MIN(salary), MAX(salary)
   INTO  minsalary, maxsalary
   FROM  employees
   WHERE job_id = :NEW.job_id;
  IF :NEW.salary < minsalary OR
     :NEW.salary > maxsalary THEN
     RAISE_APPLICATION_ERROR(-20505,'Out of range');
  END IF;
END;
/
```

ORACLE

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 that enables you to create a trigger in any schema**
- **The `ADMINISTER DATABASE TRIGGER` privilege that 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 use the privileges of the trigger owner, not the privileges of the user executing the operation that fires the trigger.

Business Application Scenarios for Implementing Triggers

You can use triggers for:

- Security
- Auditing
- Data integrity
- Referential integrity
- Table replication
- Computing derived data automatically
- Event logging

Note: Appendix C covers each of these examples in more detail.

Viewing Trigger Information

You can view the following trigger information:

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

Using USER_TRIGGERS

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

ORACLE

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_CLAUS	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_PRES ', 'AD_VP')) AND :NE W.SAL

Summary

In this lesson, you should have learned how to:

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

Practice 11: Overview

This practice covers the following topics:

- Creating advanced triggers to manage data integrity rules
- Creating triggers that cause a mutating table exception
- Creating triggers that use package state to solve the mutating table problem

12

Understanding and Influencing the PL/SQL Compiler

ORACLE

Objectives

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

- Describe native and interpreted compilations
- List the features of native compilation
- Switch between native and interpreted compilations
- Set parameters that influence PL/SQL compilation
- Query data dictionary views on how PL/SQL code is compiled
- Use the compiler warning mechanism and the `DBMS_WARNING` package to implement compiler warnings

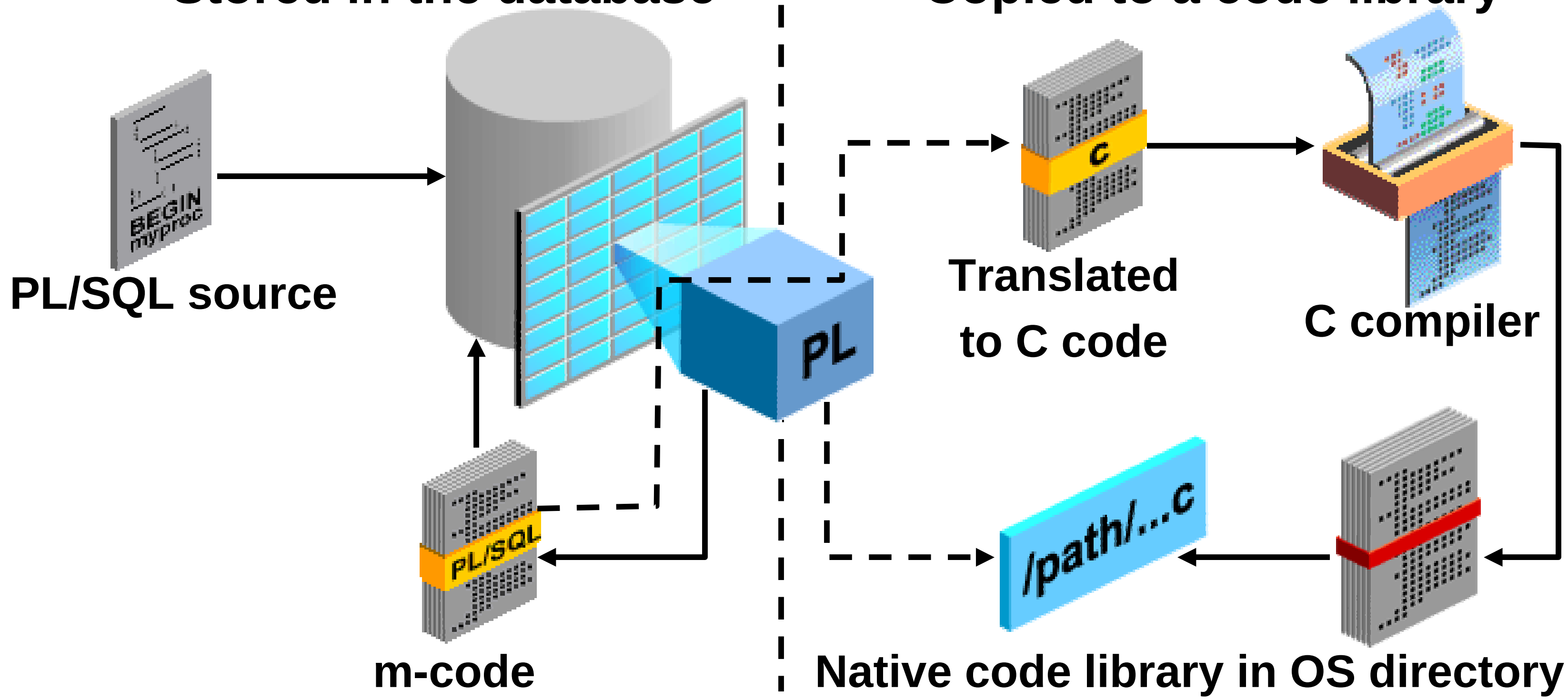
Native and Interpreted Compilation

Interpreted code

- Compiled to m-code
- Stored in the database

Natively compiled code

- Translated C and compiled
- Copied to a code library



Features and Benefits of Native Compilation

Native compilation:

- Uses a generic `makefile` that uses the following operating system software:
  - C compiler
  - Linker
  - Make utility
- Generates shared libraries that are copied to the file system and loaded at run time
- Provides better performance (up to 30% faster than interpreted code) for computation-intensive procedural operations

Considerations When Using Native Compilation

Consider the following:

- Debugging tools for PL/SQL cannot debug natively compiled code.
- Natively compiled code is slower to compile than interpreted code.
- Large amounts of natively compiled subprograms can affect performance due to operating system–imposed limitations when handling shared libraries. OS directory limitations can be managed by setting database initialization parameters:
  - `PLSQL_NATIVE_LIBRARY_SUBDIR_COUNT` and
  - `PLSQL_NATIVE_LIBRARY_DIR`

Parameters Influencing Compilation

System parameters are set in the `init`*`SID`*`.ora` file or by using the `SPFILE`:

```
PLSQL_NATIVE_LIBRARY_DIR = full-directory-path-name
PLSQL_NATIVE_LIBRARY_SUBDIR_COUNT = count
```

System or session parameters

```
PLSQL_COMPILER_FLAGS = 'NATIVE' or 'INTERPRETED'
```

ORACLE

Switching Between Native and Interpreted Compilation

- **Setting native compilation:**
  - **For the system:**

```
ALTER SYSTEM SET plsql_compiler_flags='NATIVE';
```

  - **For the session:**

```
ALTER SESSION SET plsql_compiler_flags='NATIVE';
```

- **Setting interpreted compilation:**
  - **For the system level:**

```
ALTER SYSTEM
         SET plsql_compiler_flags='INTERPRETED';
```

  - **For the session:**

```
ALTER SESSION
         SET plsql_compiler_flags='INTERPRETED';
```

Viewing Compilation Information in the Data Dictionary

Query information in the following views:

- **`USER_STORED_SETTINGS`**
- **`USER_PLSQL_OBJECTS`**

Example:

```
SELECT param_value
FROM   user_stored_settings
WHERE  param_name = 'plsql_compiler_flags'
  AND  object_name = 'GET_EMPLOYEES';
```

Note: The `PARAM_VALUE` column has a value of `NATIVE` for procedures that are compiled for native execution; otherwise, it has a value of `INTERPRETED`.

Using Native Compilation

To enable native compilation, perform the following steps:

1. Edit the supplied `makefile` and enter appropriate paths and other values for your system.
2. Set the `PLSQL_COMPILER_FLAGS` parameter (at system or session level) to the value `NATIVE`. The default is `INTERPRETED`.
3. Compile the procedures, functions, and packages.
4. Query the data dictionary to see that a procedure is compiled for native execution.

ORACLE

Compiler Warning Infrastructure

The PL/SQL compiler in Oracle Database 10*g* has been enhanced to produce warnings for subprograms. Warning levels:

- Can be set:
  - Declaratively with the `PLSQL_WARNINGS` initialization parameter
  - Programmatically using the `DBMS_WARNINGS` package
- Are arranged in three categories: severe, performance, and informational
- Can be enabled and disabled by category or a specific message

Examples of warning messages:

SP2-0804: Procedure created with compilation warnings

PLW-07203: Parameter `'IO_TBL'` may benefit from use of the `NOCOPY` compiler hint.

Setting Compiler Warning Levels

Set the `PLSQL_WARNINGS` initialization parameter to enable the database to issue warning messages.

```
ALTER SESSION SET PLSQL_WARNINGS = 'ENABLE:SEVERE',
                        'DISABLE:INFORMATIONAL';
```

- The `PLSQL_WARNINGS` combine a qualifier value (`ENABLE`, `DISABLE`, or `ERROR`) with a comma-separated list of message numbers, or with one of the following modifier values:
  - `ALL`, `SEVERE`, `INFORMATIONAL`, or `PERFORMANCE`
- Warning messages use a `PLW` prefix.

  PLW-07203: Parameter `'IO_TBL'` may benefit from use of the `NOCOPY` compiler hint.

ORACLE

Guidelines for Using PLSQL_WARNINGS

The `PLSQL_WARNINGS` setting:

- Can be set to `DEFERRED` at the system level
- Is stored with each compiled subprogram
- That is current for the session is used, by default, when recompiling with:
  - A `CREATE OR REPLACE` statement
  - An `ALTER...COMPILE` statement
- That is stored with the compiled subprogram is used when `REUSE SETTINGS` is specified when recompiling with an `ALTER...COMPILE` statement

DBMS_WARNING Package

The DBMS_WARNING package provides a way to programmatically manipulate the behavior of current system or session PL/SQL warning settings. Using DBMS_WARNING subprograms, you can:

- Query existing settings
- Modify the settings for specific requirements or restore original settings
- Delete the settings

Example: Saving and restoring warning settings for a development environment that calls your code that compiles PL/SQL subprograms, and suppresses warnings due to business requirements

Using DBMS_WARNING Procedures

- Package procedures change PL/SQL warnings:

```
ADD_WARNING_SETTING_CAT(w_category,w_value,scope)
ADD_WARNING_SETTING_NUM(w_number,w_value,scope)
SET_WARNING_SETTING_STRING(w_value, scope)
```

  - All parameters are IN parameters and have the VARCHAR2 data type. However, the w_number parameter is a NUMBER data type.
  - Parameter string values are not case sensitive.
  - The w_value parameters values are ENABLE, DISABLE, and ERROR.
  - The w_category values are ALL, INFORMATIONAL, SEVERE, and PERFORMANCE.
  - The scope value is either SESSION or SYSTEM. Using SYSTEM requires the ALTER SYSTEM privilege.

Using DBMS_WARNING Functions

- **Package functions read PL/SQL warnings:**

```
GET_CATEGORY(w_number) RETURN VARCHAR2
GET_WARNING_SETTING_CAT(w_category)RETURN VARCHAR2
GET_WARNING_SETTING_NUM(w_number) RETURN VARCHAR2
GET_WARNING_SETTING_STRING RETURN VARCHAR2
```

  – **`GET_CATEGORY` returns a value of `ALL`, `INFORMATIONAL`, `SEVERE`, or `PERFORMANCE` for a given message number.**
  – **`GET_WARNING_SETTING_CAT` returns `ENABLE`, `DISABLE`, or `ERROR` as the current warning value for a category name, and `GET_WARNING_SETTING_NUM` returns the value for a specific message number.**
  – **`GET_WARNING_SETTING_STRING` returns the entire warning string for the current session.**

ORACLE

Using `DBMS_WARNING`: Example

Consider the following scenario:
Save current warning settings, disable warnings for the `PERFORMANCE` category, compile a PL/SQL package, and restore the original warning setting.

```
CREATE PROCEDURE compile(pkg_name VARCHAR2) IS
  warn_value VARCHAR2(200);
  compile_stmt VARCHAR2(200) :=
    'ALTER PACKAGE '|| pkg_name ||' COMPILE';
BEGIN
  warn_value :=   -- Save current settings
      DBMS_WARNING.GET_WARNING_SETTING_STRING;
  DBMS_WARNING.ADD_WARNING_SETTING_CAT( -- change
      'PERFORMANCE', 'DISABLE', 'SESSION');
  EXECUTE IMMEDIATE compile_stmt;
  DBMS_WARNING.SET_WARNING_SETTING_STRING(--restore
      warn_value, 'SESSION');
END;
```

Using DBMS_WARNING: Example

To test the `compile` procedure, you can use the following script sequence in *i*SQL*Plus:

```
DECLARE
  PROCEDURE print(s VARCHAR2) IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE(s);
  END;
BEGIN
  print('Warning settings before: '||
         DBMS_WARNING.GET_WARNING_SETTING_STRING);
  compile('my_package');
  print('Warning settings after: '||
         DBMS_WARNING.GET_WARNING_SETTING_STRING);
END;
/
SHOW ERRORS PACKAGE MY_PACKAGE
```

Summary

In this lesson, you should have learned how to:

- Switch between native and interpreted compilations
- Set parameters that influence native compilation of PL/SQL programs
- Query data dictionary views that provide information on PL/SQL compilation settings
- Use the PL/SQL compiler warning mechanism:
  - Declaratively by setting the `PLSQL_WARNINGS` parameter
  - Programmatically using the `DBMS_WARNING` package

Practice 12: Overview

This practice covers the following topics:

- Enabling native compilation for your session and compiling a procedure
- Creating a subprogram to compile a PL/SQL procedure, function, or a package; suppressing warnings for the `PERFORMANCE` compiler warning category; and restoring the original session warning settings
- Executing the procedure to compile a PL/SQL package containing a procedure that uses a PL/SQL table as an `IN OUT` parameter without specifying the `NOCOPY` hint

C

Studies for Implementing Triggers

Objectives

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

- Enhance database security with triggers
- Audit data changes using DML triggers
- Enforce data integrity with DML triggers
- Maintain referential integrity using triggers
- Use triggers to replicate data between tables
- Use triggers to automate computation of derived data
- Provide event-logging capabilities using triggers

Controlling Security Within the Server

Using database security with the `GRANT` statement.

```
GRANT SELECT, INSERT, UPDATE, DELETE
 ON    employees
 TO    clerk;                  -- database role
GRANT clerk TO scott;
```

Controlling Security with a Database Trigger

```
CREATE OR REPLACE TRIGGER secure_emp
  BEFORE INSERT OR UPDATE OR DELETE ON employees
DECLARE
dummy PLS_INTEGER;
BEGIN
 IF (TO_CHAR (SYSDATE, 'DY') IN ('SAT','SUN')) THEN
   RAISE_APPLICATION_ERROR(-20506,'You may only
     change data during normal business hours.');
 END IF;
 SELECT COUNT(*) INTO dummy FROM holiday
 WHERE holiday_date = TRUNC (SYSDATE);
 IF dummy > 0 THEN
   RAISE_APPLICATION_ERROR(-20507,
     'You may not change data on a holiday.');
 END IF;
END;
/
```

Using the Server Facility to Audit Data Operations

The Oracle server stores the audit information in a data dictionary table or an operating system file.

```
AUDIT INSERT, UPDATE, DELETE
  ON  departments
  BY ACCESS
WHENEVER SUCCESSFUL;
```

```
Audit succeeded.
```

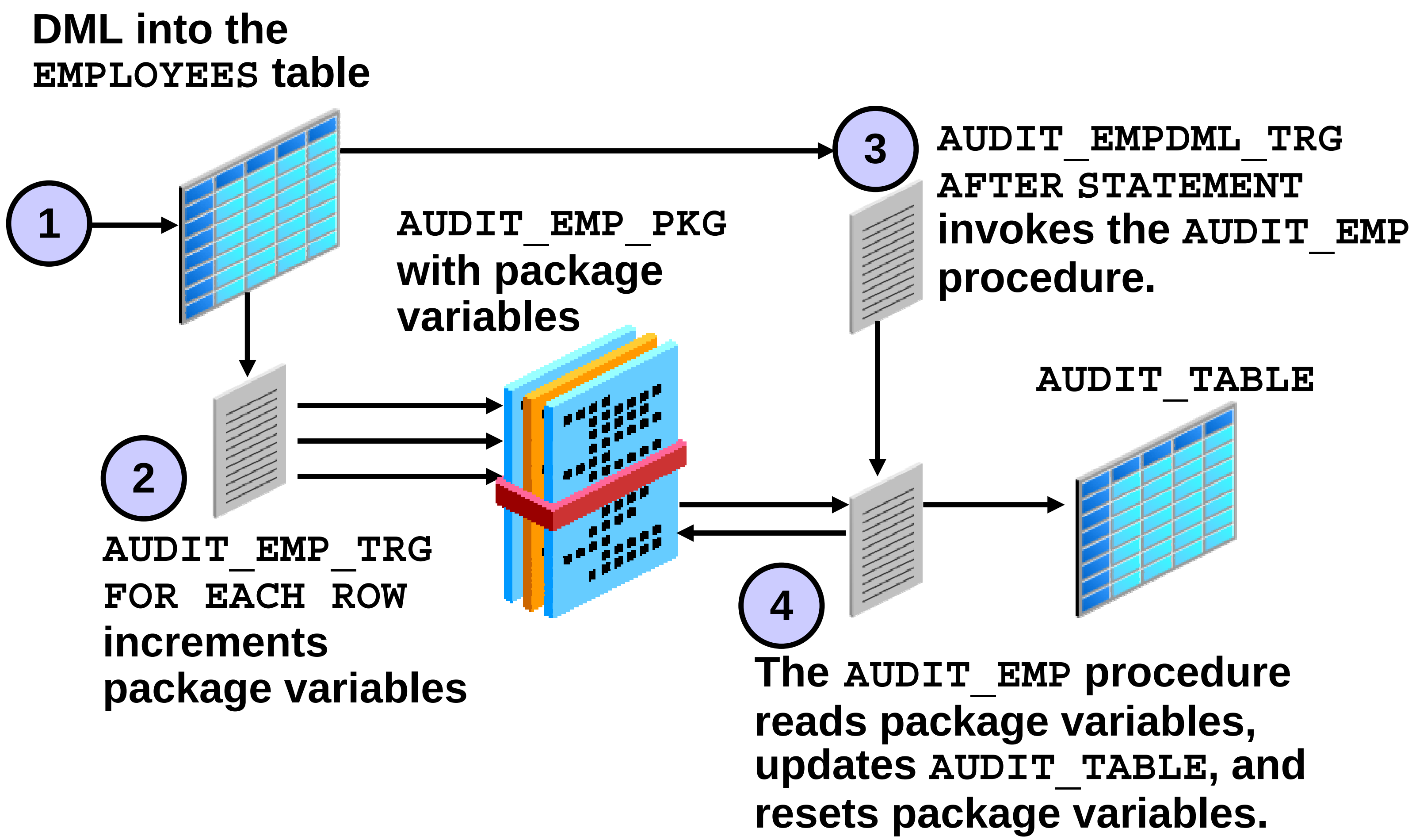
ORACLE

Auditing by Using a Trigger

```
CREATE OR REPLACE TRIGGER audit_emp_values
AFTER DELETE OR INSERT OR UPDATE
ON employees FOR EACH ROW
BEGIN
 IF (audit_emp_pkg. reason IS NULL) THEN
   RAISE_APPLICATION_ERROR (-20059, 'Specify a
    reason for operation through the procedure
    AUDIT_EMP_PKG.SET_REASON to proceed.');
 ELSE
   INSERT INTO audit_emp_table (user_name,
     timestamp, id, old_last_name, new_last_name,
     old_salary, new_salary, comments)
   VALUES (USER, SYSDATE, :OLD.employee_id,
     :OLD.last_name, :NEW.last_name,:OLD.salary,
     :NEW.salary, audit_emp_pkg.reason);
 END IF;
END;
```

```
CREATE OR REPLACE TRIGGER cleanup_audit_emp
 AFTER INSERT OR UPDATE OR DELETE ON employees
BEGIN audit_emp_package.g_reason := NULL;
END;
```

Auditing Triggers by Using Package Constructs



Auditing Triggers by Using Package Constructs

`AFTER` **statement trigger:**

```
CREATE OR REPLACE TRIGGER audit_empdml_trg
AFTER UPDATE OR INSERT OR DELETE on employees
BEGIN
  audit_emp;       -- write the audit data
END audit_emp_tab;
/
```

`AFTER` **row trigger:**

```
CREATE OR REPLACE TRIGGER audit_emp_trg
AFTER UPDATE OR INSERT OR DELETE ON EMPLOYEES
FOR EACH ROW
-- Call Audit package to maintain counts
CALL audit_emp_pkg.set(INSERTING,UPDATING,DELETING);
/
```

AUDIT_PKG Package

```
CREATE OR REPLACE PACKAGE audit_emp_pkg IS
  delcnt PLS_INTEGER := 0;
  inscnt PLS_INTEGER := 0;
  updcnt  PLS_INTEGER := 0;
  PROCEDURE init;
  PROCEDURE set(i BOOLEAN,u BOOLEAN,d BOOLEAN);
END audit_emp_pkg;
/
CREATE OR REPLACE PACKAGE BODY audit_emp_pkg IS
 PROCEDURE init IS
  BEGIN
    inscnt := 0; updcnt := 0; delcnt := 0;
  END;
  PROCEDURE set(i BOOLEAN,u BOOLEAN,d BOOLEAN) IS
  BEGIN
    IF i THEN inscnt := inscnt + 1;
    ELSIF d THEN delcnt := delcnt + 1;
    ELSE upd := updcnt + 1;
    END IF;
  END;
END audit_emp_pkg;
/
```

AUDIT_TABLE Table and AUDIT_EMP Procedure

```
CREATE TABLE audit_table (
 USER_NAME   VARCHAR2(30),
 TABLE_NAME  VARCHAR2(30),
 INS         NUMBER,
 UPD         NUMBER,
 DEL         NUMBER)
/
CREATE OR REPLACE PROCEDURE audit_emp IS
BEGIN
  IF  delcnt + inscnt + updcnt <> 0 THEN
    UPDATE audit_table
     SET del = del + audit_emp_pkg.delcnt,
         ins = ins + audit_emp_pkg.inscnt,
         upd = upd + audit_emp_pkg.updcnt
    WHERE user_name = USER
    AND   table_name = 'EMPLOYEES';
    audit_emp_pkg.init;
  END IF;
END audit_emp;
/
```

Enforcing Data Integrity Within the Server

```
ALTER TABLE employees ADD
  CONSTRAINT ck_salary CHECK (salary >= 500);
```

```
Table altered.
```

Protecting Data Integrity with a Trigger

```
CREATE OR REPLACE TRIGGER check_salary
  BEFORE UPDATE OF salary ON employees
  FOR EACH ROW
  WHEN (NEW.salary < OLD.salary)
BEGIN
  RAISE_APPLICATION_ERROR (-20508,
      'Do not decrease salary.');
END;
/
```

Enforcing Referential Integrity Within the Server

```
ALTER TABLE employees
  ADD CONSTRAINT emp_deptno_fk
  FOREIGN KEY (department_id)
     REFERENCES departments(department_id)
ON DELETE CASCADE;
```

Protecting Referential Integrity with a Trigger

```
CREATE OR REPLACE TRIGGER cascade_updates
 AFTER UPDATE OF department_id ON departments
 FOR EACH ROW
BEGIN
 UPDATE employees
  SET employees.department_id=:NEW.department_id
  WHERE employees.department_id=:OLD.department_id;
 UPDATE job_history
  SET department_id=:NEW.department_id
  WHERE department_id=:OLD.department_id;
END;
/
```

Replicating a Table Within the Server

```
CREATE MATERIALIZED VIEW emp_copy
  NEXT sysdate + 7
  AS SELECT * FROM employees@ny;
```

Replicating a Table with a Trigger

```
CREATE OR REPLACE TRIGGER emp_replica
 BEFORE INSERT OR UPDATE ON employees FOR EACH ROW
BEGIN /* Proceed if user initiates data operation,
        NOT through the cascading trigger.*/
  IF INSERTING THEN
   IF :NEW.flag IS NULL THEN
     INSERT INTO employees@sf
     VALUES(:new.employee_id,...,'B');
     :NEW.flag := 'A';
   END IF;
  ELSE   /* Updating. */
   IF :NEW.flag = :OLD.flag THEN
     UPDATE employees@sf
      SET ename=:NEW.last_name,...,flag=:NEW.flag
      WHERE employee_id = :NEW.employee_id;
   END IF;
   IF :OLD.flag = 'A' THEN :NEW.flag := 'B';
                      ELSE :NEW.flag := 'A';
   END IF;
  END IF;
END;
```

ORACLE

Computing Derived Data Within the Server

```
UPDATE departments
 SET total_sal=(SELECT SUM(salary)
                FROM employees
                WHERE employees.department_id =
                    departments.department_id);
```

Computing Derived Values with a Trigger

```
CREATE PROCEDURE increment_salary
  (id NUMBER, new_sal NUMBER) IS
BEGIN
   UPDATE departments
   SET    total_sal = NVL (total_sal, 0)+ new_sal
   WHERE  department_id = id;
END increment_salary;
```

```
CREATE OR REPLACE TRIGGER compute_salary
AFTER INSERT OR UPDATE OF salary OR DELETE
ON employees FOR EACH ROW
BEGIN
 IF DELETING THEN     increment_salary(
     :OLD.department_id,(-1*:OLD.salary));
 ELSIF UPDATING THEN  increment_salary(
     :NEW.department_id,(:NEW.salary-:OLD.salary));
 ELSE   increment_salary(
     :NEW.department_id,:NEW.salary); --INSERT
 END IF;
END;
```

Logging Events with a Trigger

```
CREATE OR REPLACE TRIGGER notify_reorder_rep
BEFORE UPDATE OF quantity_on_hand, reorder_point
 ON inventories FOR EACH ROW
DECLARE
 dsc product_descriptions.product_description%TYPE;
 msg_text VARCHAR2(2000);
BEGIN
  IF :NEW.quantity_on_hand <=
     :NEW.reorder_point THEN
    SELECT product_description INTO dsc
    FROM product_descriptions
    WHERE product_id = :NEW.product_id;
  _ msg_text := 'ALERT: INVENTORY LOW ORDER:'||
       'Yours,' ||CHR(10) ||user || '.'|| CHR(10);
  ELSIF :OLD.quantity_on_hand >=
        :NEW.quantity_on_hand THEN
    msg_text := 'Product #'||... CHR(10);
  END IF;
  UTL_MAIL.SEND('inv@oracle.com','ord@oracle.com',
   message=>msg_text, subject=>'Inventory Notice');
END;
```

Summary

In this lesson, you should have learned how to:

- Use database triggers and database server functionality to:
  - Enhance database security
  - Audit data changes
  - Enforce data integrity
  - Maintain referential integrity
  - Replicate data between tables
  - Automate computation of derived data
  - Provide event-logging capabilities
- Recognize when to use triggers to database functionality

ORACLE

D

Review of PL/SQL

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)

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;
  v_count          BINARY_INTEGER := 0;
  v_valid          BOOLEAN NOT NULL := TRUE;
```

ORACLE

Declaring Variables with the %TYPE Attribute

Examples:

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

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
    (ename    VARCHAR2(25),
     job          VARCHAR2(10),
     sal          NUMBER(8,2));
  emp_record     emp_record_type;
...
```

`%ROWTYPE` Attribute

Examples:

- **Declare a variable to store the same information about a department as is stored in the `DEPARTMENTS` table.**

```
dept_record    departments%ROWTYPE;
```

- **Declare a variable to store the same information about an employee as is stored in the `EMPLOYEES` table.**

```
emp_record     employees%ROWTYPE;
```

Creating a PL/SQL 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;
```

`SELECT` Statements in PL/SQL

The `INTO` clause is mandatory.

Example:

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

Inserting Data

Add new employee information to the `EMPLOYEES` table.

Example:

```
DECLARE
  v_empid  employees.employee_id%TYPE;
BEGIN
  SELECT  employees_seq.NEXTVAL
  INTO    v_empno
  FROM    dual;
  INSERT INTO  employees(employee_id, last_name,
                         job_id, department_id)
  VALUES(v_empno, 'HARDING', 'PU_CLERK', 30);
END;
```

Updating Data

Increase the salary of all employees in the `EMPLOYEES` table who are purchasing clerks.

Example:

```
DECLARE
  v_sal_increase   employees.salary%TYPE := 2000;
BEGIN
  UPDATE  employees
  SET     salary = salary + v_sal_increase
  WHERE   job_id = 'PU_CLERK';
END;
```

Deleting Data

Delete rows that belong to department 190 from the `EMPLOYEES` table.

Example:

```
DECLARE
  v_deptid   employees.department_id%TYPE := 190;
BEGIN
  DELETE FROM employees
  WHERE department_id = v_deptid;
END;
```

`COMMIT` and `ROLLBACK` Statements

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

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`	Boolean attribute that always evaluates to `FALSE` because PL/SQL closes implicit cursors immediately after they are executed

`IF`, `THEN`, and `ELSIF` Statements

For a given value entered, return a calculated value.

Example:

```
. . .
IF v_start > 100 THEN
  v_start := 2 * v_start;
ELSIF v_start >= 50 THEN
  v_start := 0.5 * v_start;
ELSE
  v_start := 0.1 * v_start;
END IF;
. . .
```

ORACLE

Basic Loop

Example:

```
DECLARE
  v_ordid     order_items.order_id%TYPE := 101;
  v_counter   NUMBER(2) := 1;
BEGIN
  LOOP
    INSERT INTO order_items(order_id,line_item_id)
    VALUES(v_ordid, v_counter);
    v_counter := v_counter + 1;
    EXIT WHEN v_counter > 10;
  END LOOP;
END;
```

FOR Loop

Insert the first 10 new line items for order number 101.

Example:

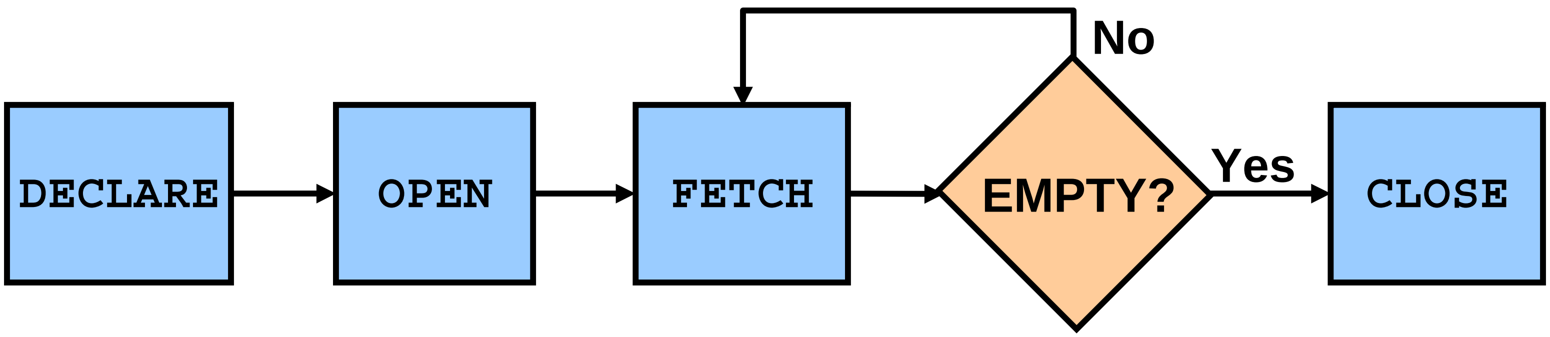
```
DECLARE
  v_ordid     order_items.order_id%TYPE := 101;
BEGIN
  FOR i IN 1..10 LOOP
    INSERT INTO order_items(order_id,line_item_id)
    VALUES(v_ordid, i);
  END LOOP;
END;
```

WHILE Loop

Example:

```
ACCEPT p_price PROMPT 'Enter the price of the item: '
ACCEPT p_itemtot -
 PROMPT 'Enter the maximum total for purchase of item: '
DECLARE
...
v_qty                      NUMBER(8) := 1;
v_running_total            NUMBER(7,2) := 0;
BEGIN
  ...
  WHILE v_running_total < &p_itemtot LOOP
    ...
  v_qty := v_qty + 1;
  v_running_total := v_qty * &p_price;
  END LOOP;
...
```

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.

Declaring the Cursor

Example:

```
DECLARE
  CURSOR c1 IS
    SELECT employee_id, last_name
    FROM   employees;

  CURSOR c2 IS
    SELECT *
    FROM   departments
    WHERE  department_id = 10;
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

Examples:

```
FETCH c1 INTO v_empid, v_ename;
```

```
...
OPEN defined_cursor;
LOOP
  FETCH defined_cursor INTO defined_variables
  EXIT WHEN ...;
  ...
    -- Process the retrieved data
  ...
END;
```

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.**

ORACLE

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

Cursor FOR Loops

Retrieve employees one by one until there are no more left.

Example:

```
DECLARE
  CURSOR c1 IS
    SELECT employee_id, last_name
    FROM   employees;
BEGIN
  FOR emp_record IN c1 LOOP
         -- implicit open and implicit fetch occur
    IF emp_record.employee_id = 134 THEN
      ...
  END LOOP; -- implicit close occurs
END;
```

FOR UPDATE Clause

Retrieve the orders for amounts over $1,000 that were processed today.

Example:

```
DECLARE
  CURSOR c1 IS
    SELECT customer_id, order_id
    FROM   orders
    WHERE  order_date = SYSDATE
      AND  order_total > 1000.00
    ORDER BY customer_id
    FOR UPDATE NOWAIT;
```

WHERE CURRENT OF Clause

Example:

```
DECLARE
  CURSOR c1 IS
    SELECT salary FROM employees
    FOR UPDATE OF salary NOWAIT;
BEGIN
  ...
  FOR emp_record IN c1 LOOP
    UPDATE ...
      WHERE CURRENT OF c1;
    ...
  END LOOP;
  COMMIT;
END;
```

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`

Trapping Predefined Oracle Server Errors: Example

Syntax:

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

ORACLE

Non-Predefined Error

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

```
DECLARE
  e_products_invalid EXCEPTION;
  PRAGMA EXCEPTION_INIT (
              e_products_invalid, -2292);
  v_message VARCHAR2(50);
BEGIN
. . .
EXCEPTION
  WHEN e_products_invalid THEN
    :g_message := 'Product ID
              specified is not valid.';
. . .
END;
```

User-Defined Exceptions

Example:

```
[DECLARE]
  e_amount_remaining EXCEPTION;
. . .
BEGIN
. . .
  RAISE e_amount_remaining;
. . .
EXCEPTION
  WHEN e_amount_remaining THEN
    :g_message := 'There is still an amount
                   in stock.';
. . .
END;
```

1

2

3

`RAISE_APPLICATION_ERROR` Procedure

Syntax:

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

- **Enables you to issue user-defined error messages from stored subprograms**
- **Is called from an executing stored subprogram only**

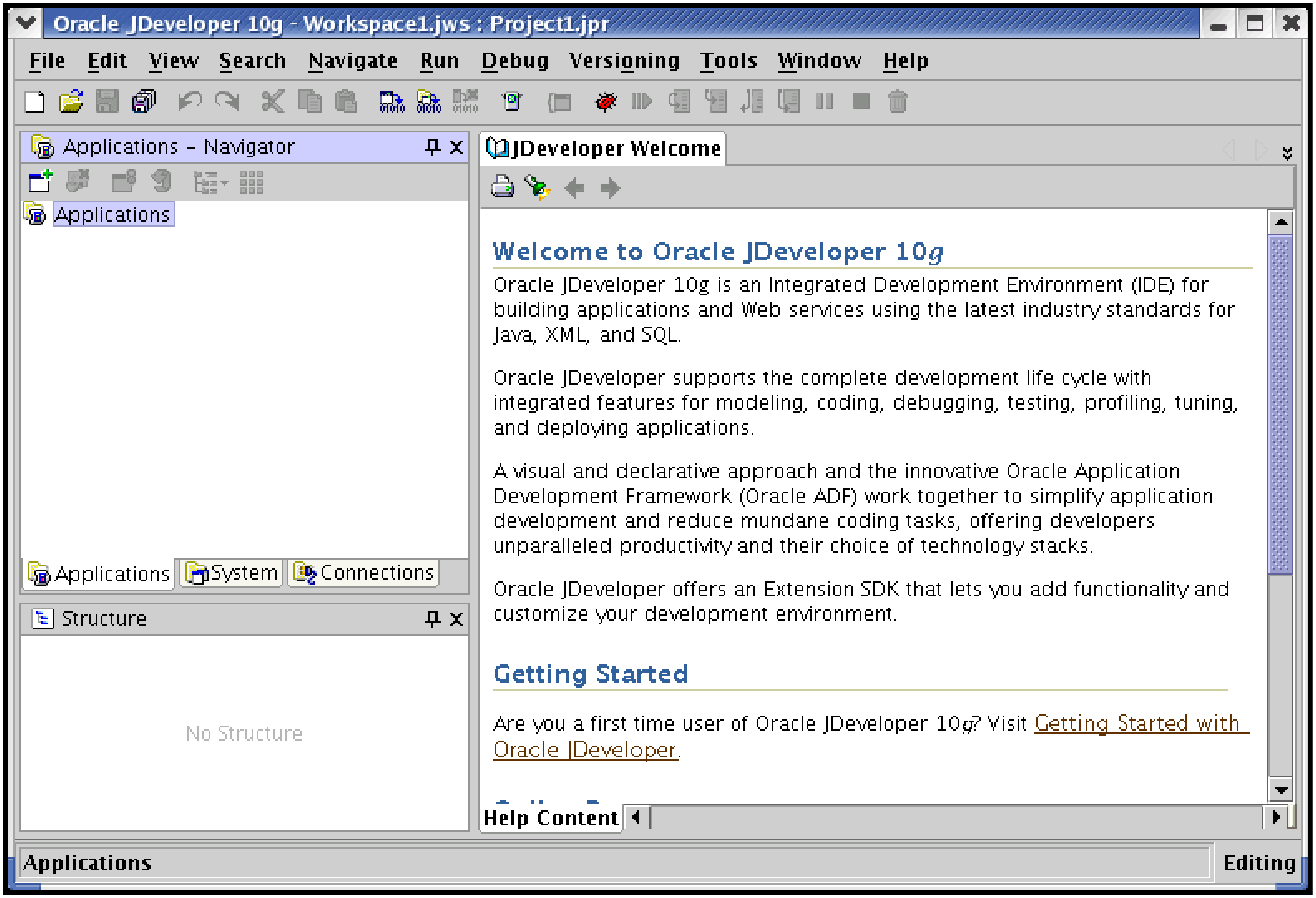
`RAISE_APPLICATION_ERROR` Procedure

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

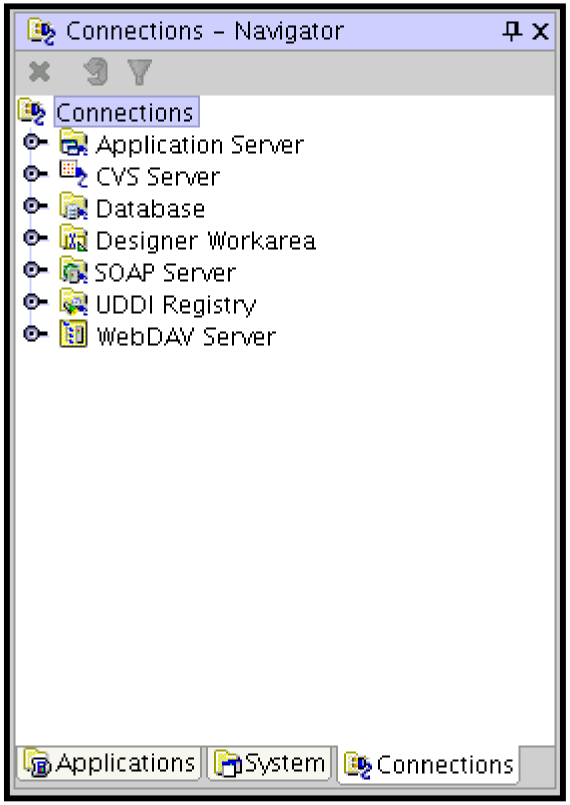
E

JDeveloper

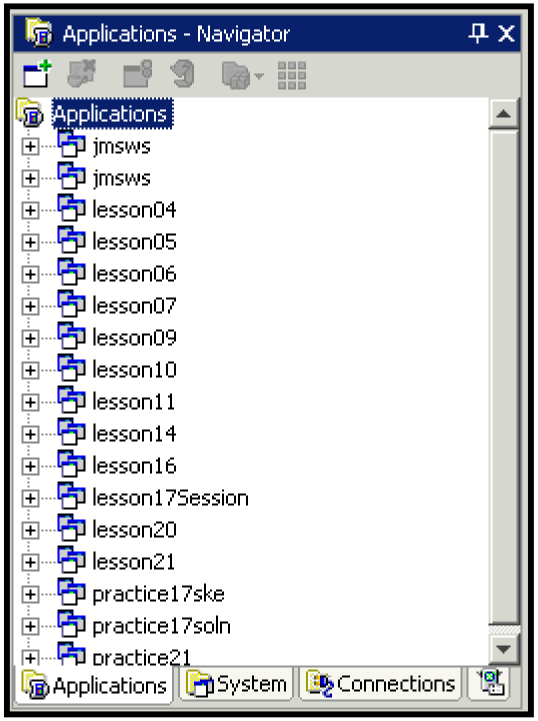
JDeveloper


Oracle JDeveloper 10g - Workspace1.jws : Project1.jpr
File Edit View Search Navigate Run Debug Versioning Tools Window Help
Applications – Navigator
Applications
Applications System Connections
Structure
No Structure
JDeveloper Welcome
Welcome to Oracle JDeveloper 10g
Oracle JDeveloper 10g is an Integrated Development Environment (IDE) for building applications and Web services using the latest industry standards for Java, XML, and SQL.
Oracle JDeveloper supports the complete development life cycle with integrated features for modeling, coding, debugging, testing, profiling, tuning, and deploying applications.
A visual and declarative approach and the innovative Oracle Application Development Framework (Oracle ADF) work together to simplify application development and reduce mundane coding tasks, offering developers unparalleled productivity and their choice of technology stacks.
Oracle JDeveloper offers an Extension SDK that lets you add functionality and customize your development environment.
Getting Started
Are you a first time user of Oracle JDeveloper 10g? Visit Getting Started with Oracle JDeveloper.
Help Content
Applications
Editing

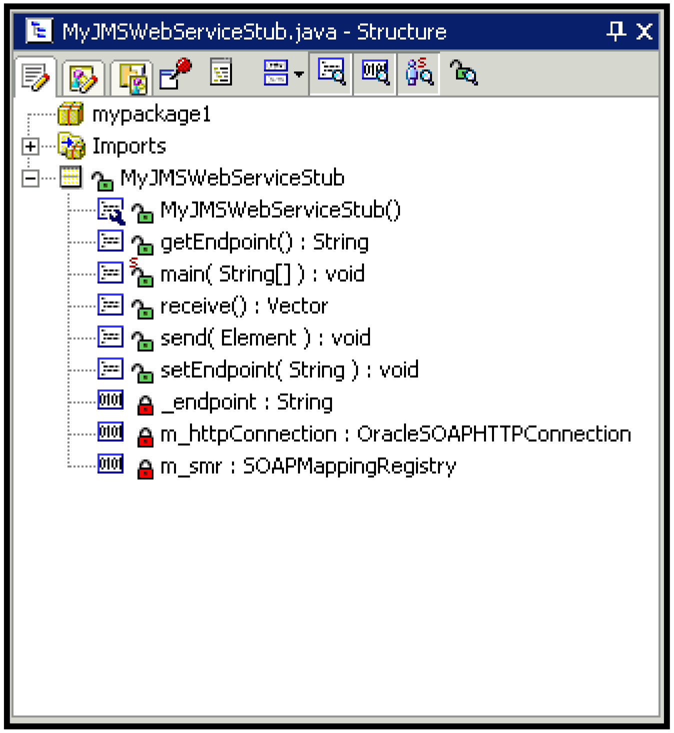
Connection Navigator



Application Navigator



Structure Window



Editor Window

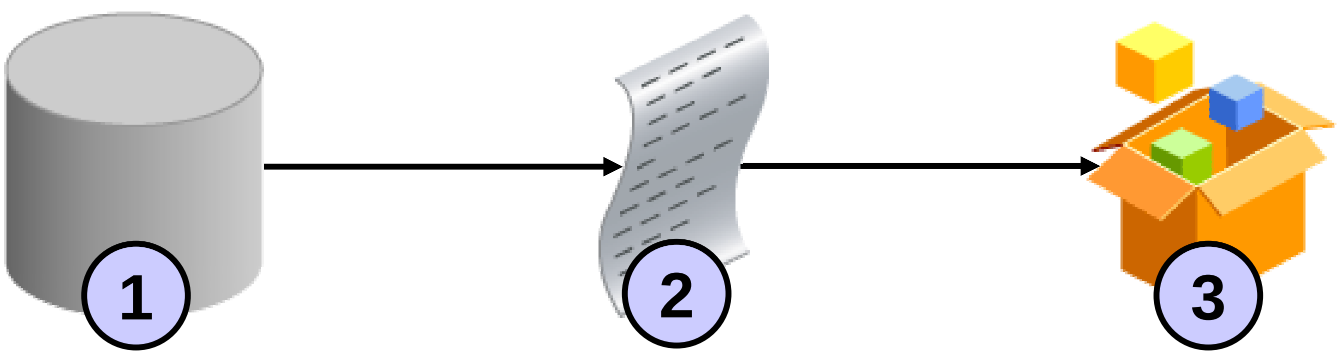
SHOW_CUST_CALL

```
PROCEDURE show_cust_call (
custid IN NUMBER default 101) AS
 BEGIN NULL;
htp.prn('
');
htp.prn('
');
htp.prn('
<HTML>
<BODY>
<form method="POST" action="show_cust">
<p>Enter the Customer ID:
<input type="text" name="custid">
<input type="submit" value="Submit">
</form>
</BODY>
</HTML>
');
 END;
```

Deploying Java Stored Procedures

Before deploying Java stored procedures, perform the following steps:

1. Create a database connection.
2. Create a deployment profile.
3. Deploy the objects.



Publishing Java to PL/SQL

FormatCreditCardNo.java | CCFORMAT

```java
public class FormatCreditCardNo
{
  public static final void formatCard(String[] cardno)
  {
  int count=0, space=0;
  String oldcc=cardno[0];
 // System.out.println("Printing the card no initially "+oldcc);
  String[] newcc= {""};
  while (count<16)
  {
  newcc[0]+= oldcc.charAt(count);
  space++;
  if (space ==4)
  {  newcc[0]+=" "; space=0;  }
  count++;
  }
  cardno[0]=newcc [0];
  }
}
```

FormatCreditCardNo.java | CCFORMAT

```sql
PROCEDURE ccformat (x IN OUT varchar2)
AS LANGUAGE JAVA
NAME 'FormatCreditCardNo.formatCard(java.lang.String[])';
```

ORACLE

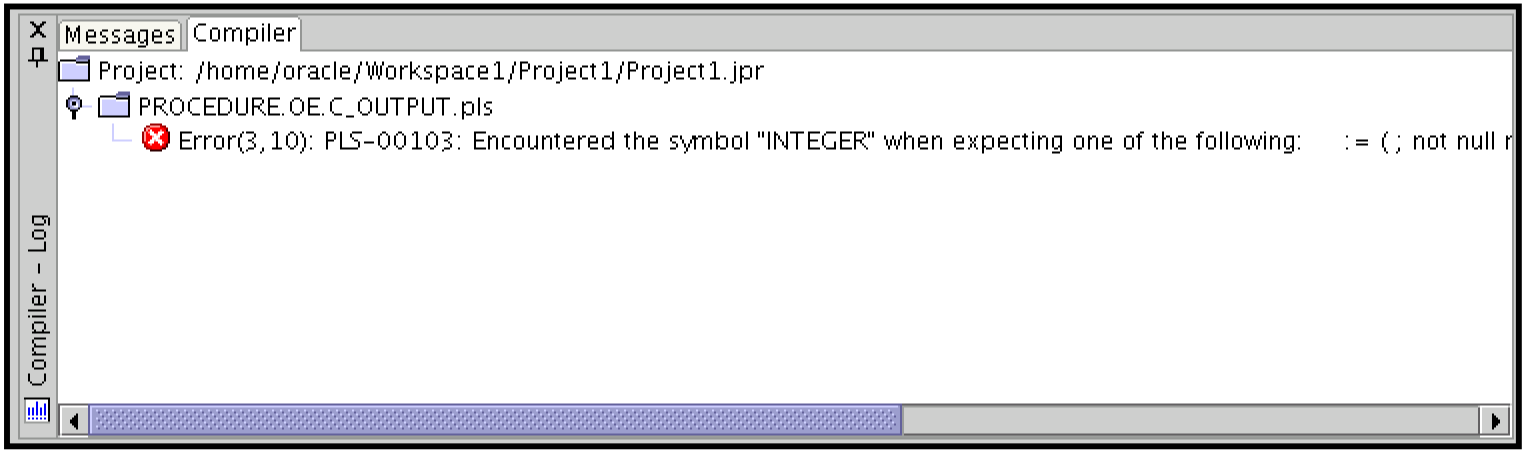
Creating Program Units

TEST_JDEV

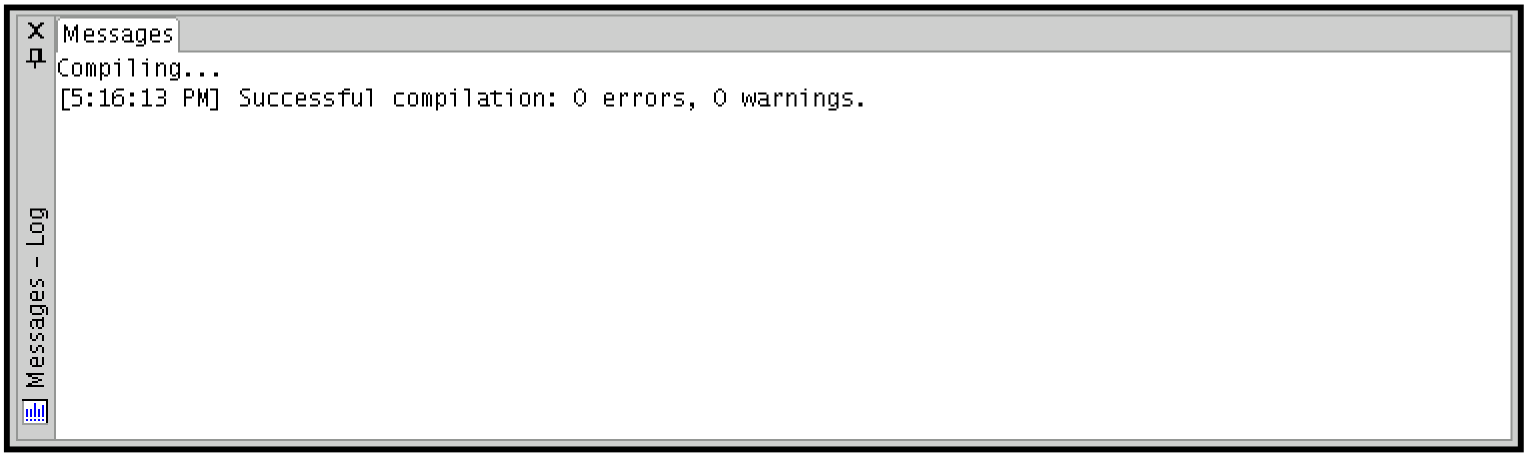
```
FUNCTION "TEST_JDEV" RETURN VARCHAR2
AS
BEGIN
  RETURN('');
END;
```

Skeleton of the function

Compiling

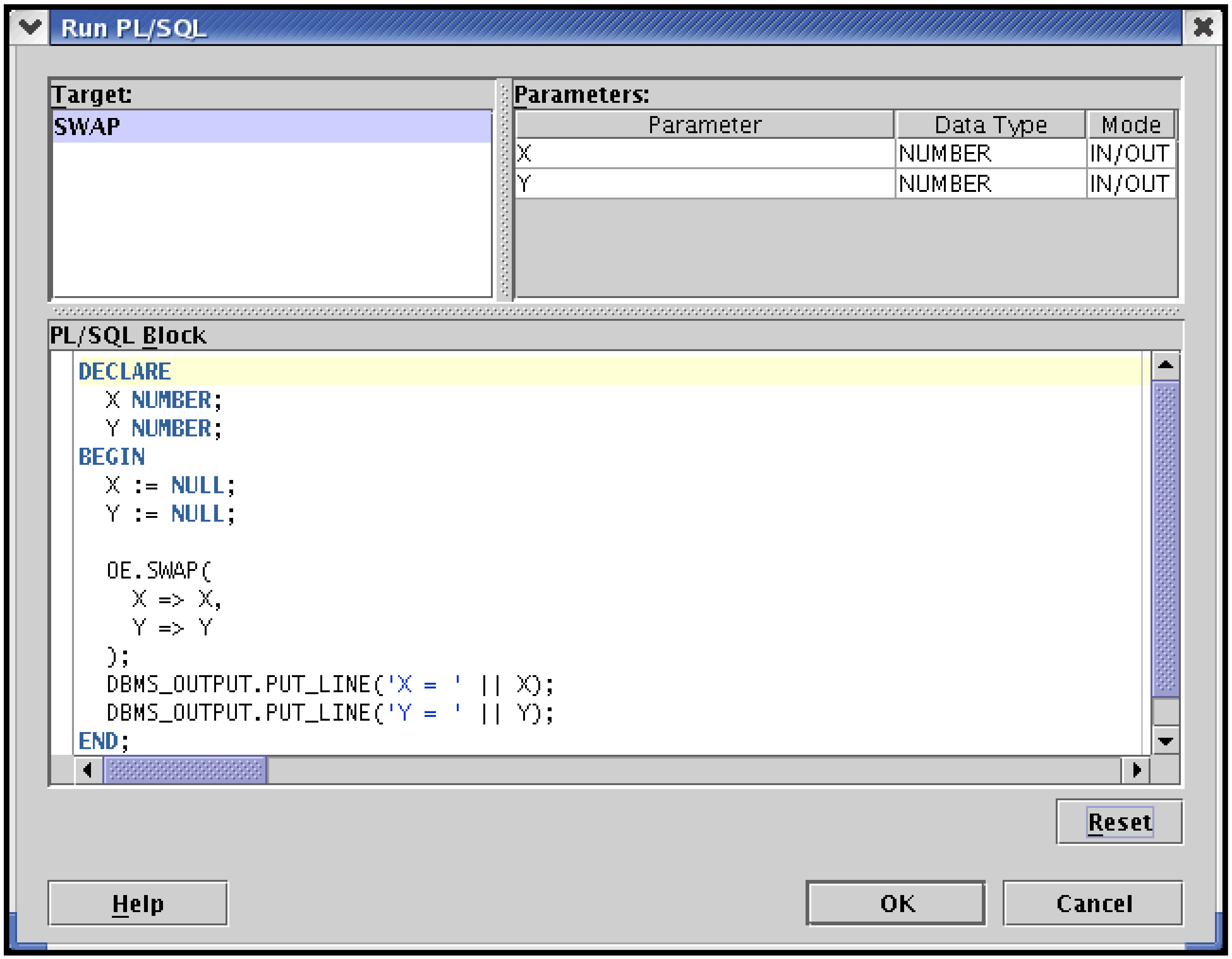


Compilation with errors



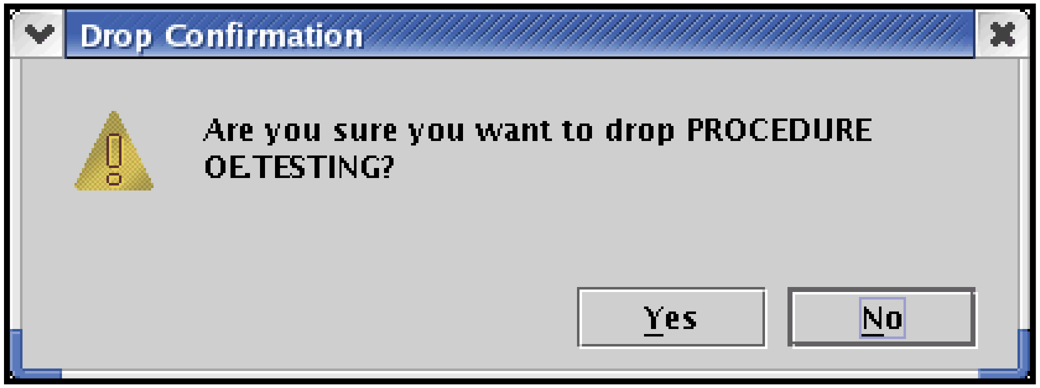
Compilation without errors

Running a Program Unit



ORACLE

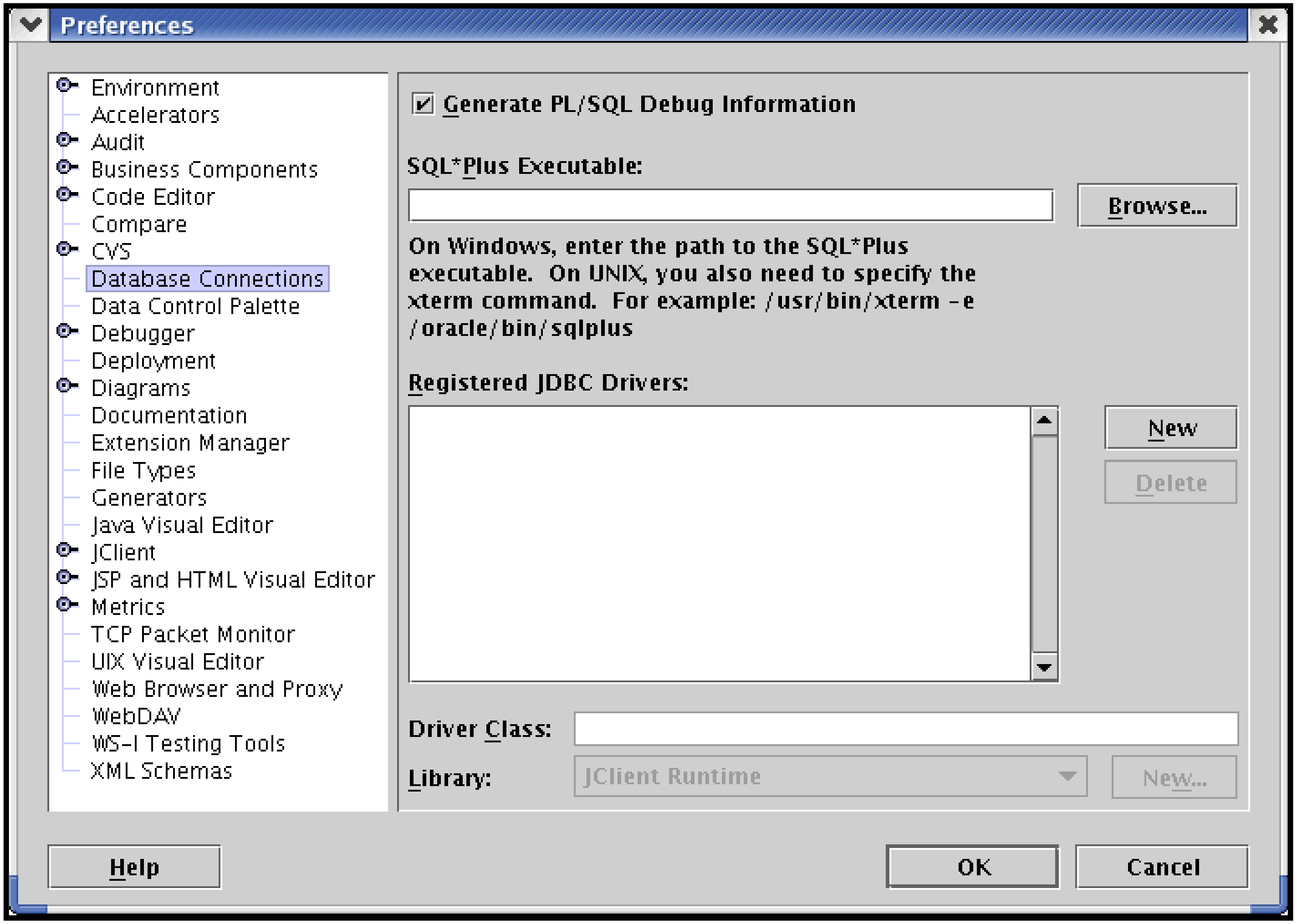
Dropping a Program Unit



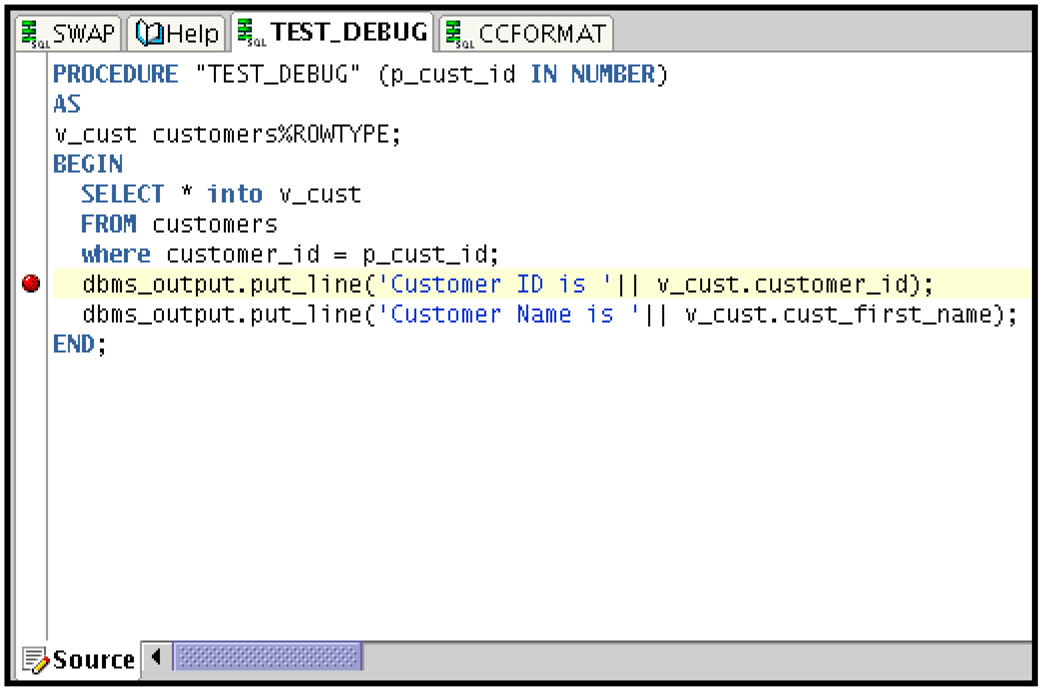
Debugging PL/SQL Programs

- JDeveloper support two types of debugging:
  - Local
  - Remote
- You need the following privileges to perform PL/SQL debugging:
  - `DEBUG ANY PROCEDURE`
  - `DEBUG CONNECT SESSION`

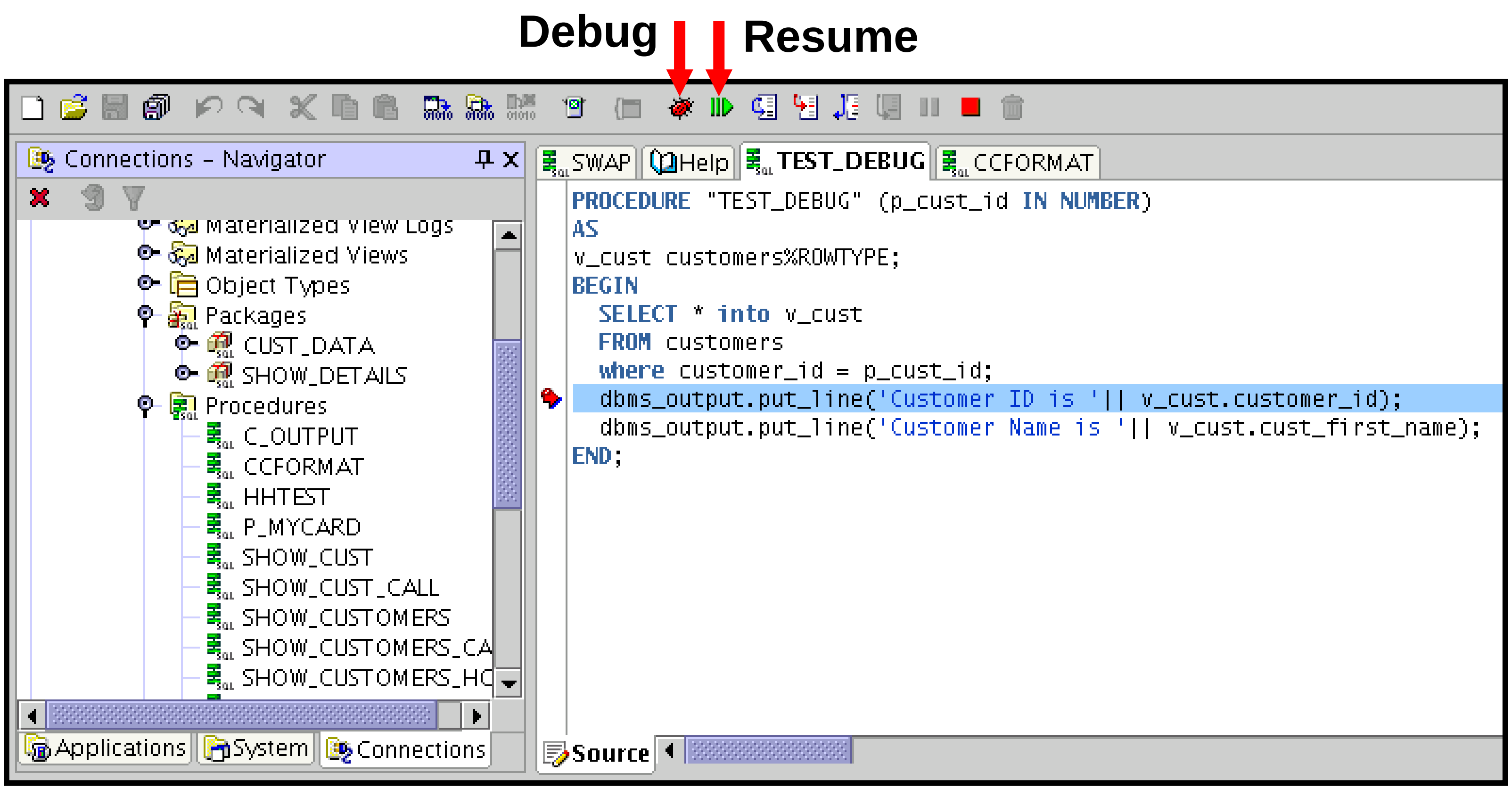
Debugging PL/SQL Programs



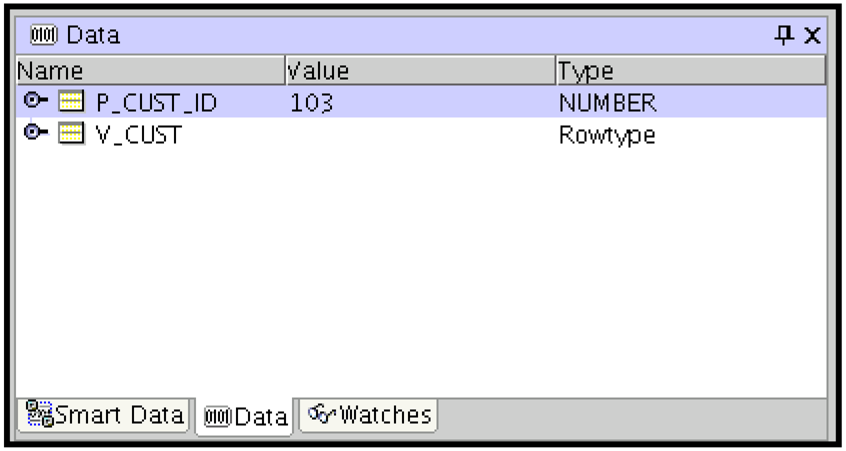
Setting Breakpoints



Stepping Through Code

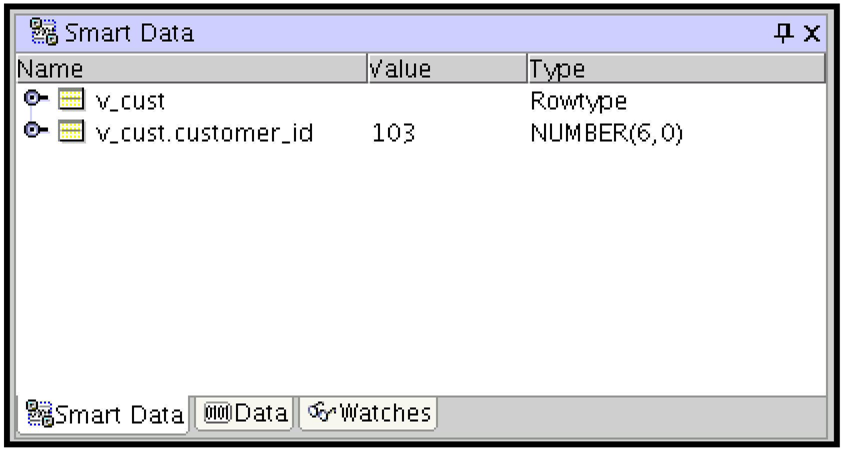


Examining and Modifying Variables



Data window

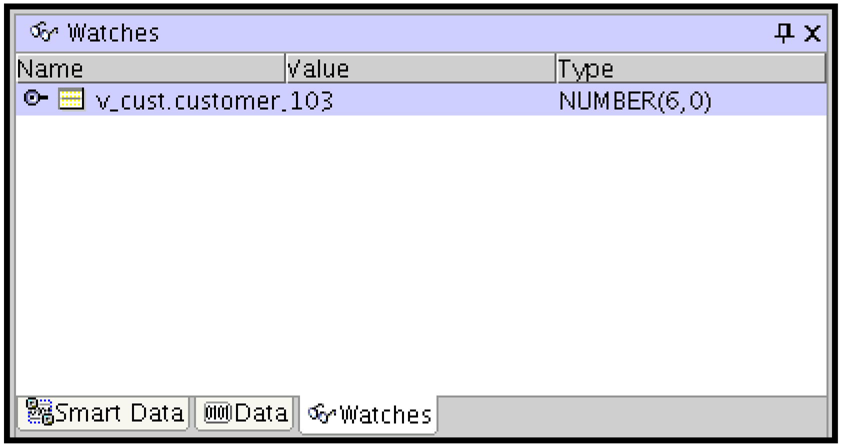
Examining and Modifying Variables



Smart Data window

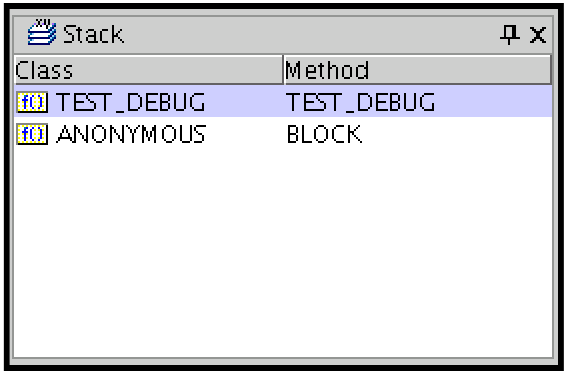
ORACLE

Examining and Modifying Variables



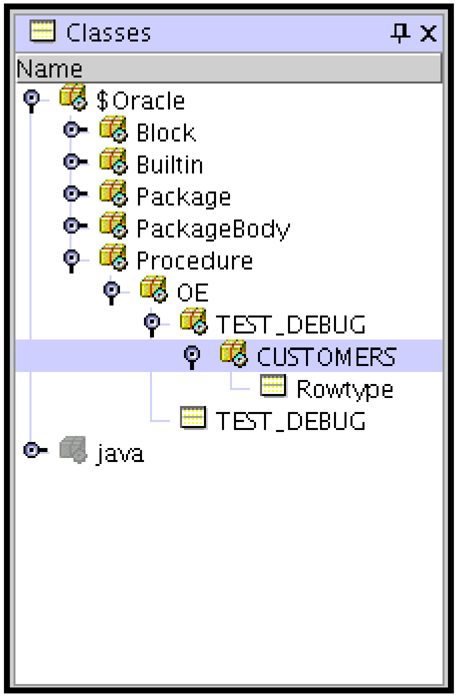
Watches window

Examining and Modifying Variables



Stack window

Examining and Modifying Variables



Classes window

F

Using SQL Developer

ORACLE

Objectives

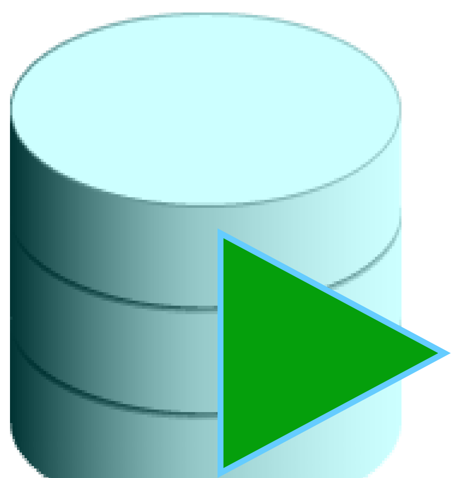
After completing this appendix, you should be able to do the following:

- List the key features of Oracle SQL Developer
- Install Oracle SQL Developer
- Identify menu items of Oracle SQL Developer
- Create a database connection
- Manage database objects
- Use the SQL Worksheet
- Execute SQL statements and SQL scripts
- Edit and debug PL/SQL statements
- Create and save reports

ORACLE

What Is Oracle SQL Developer?

- **Oracle SQL Developer is a graphical tool that enhances productivity and simplifies database development tasks.**
- **You can connect to any target Oracle database schema by using standard Oracle database authentication.**



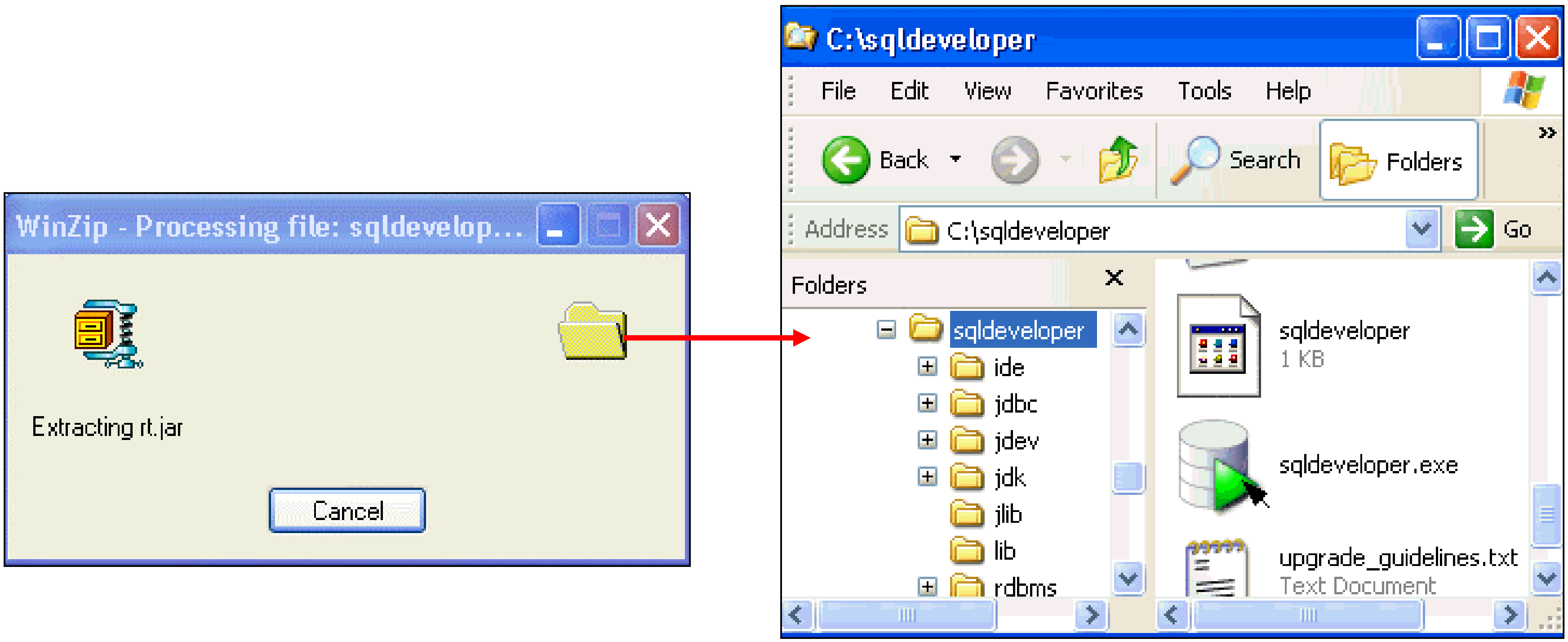
SQL Developer

Key Features

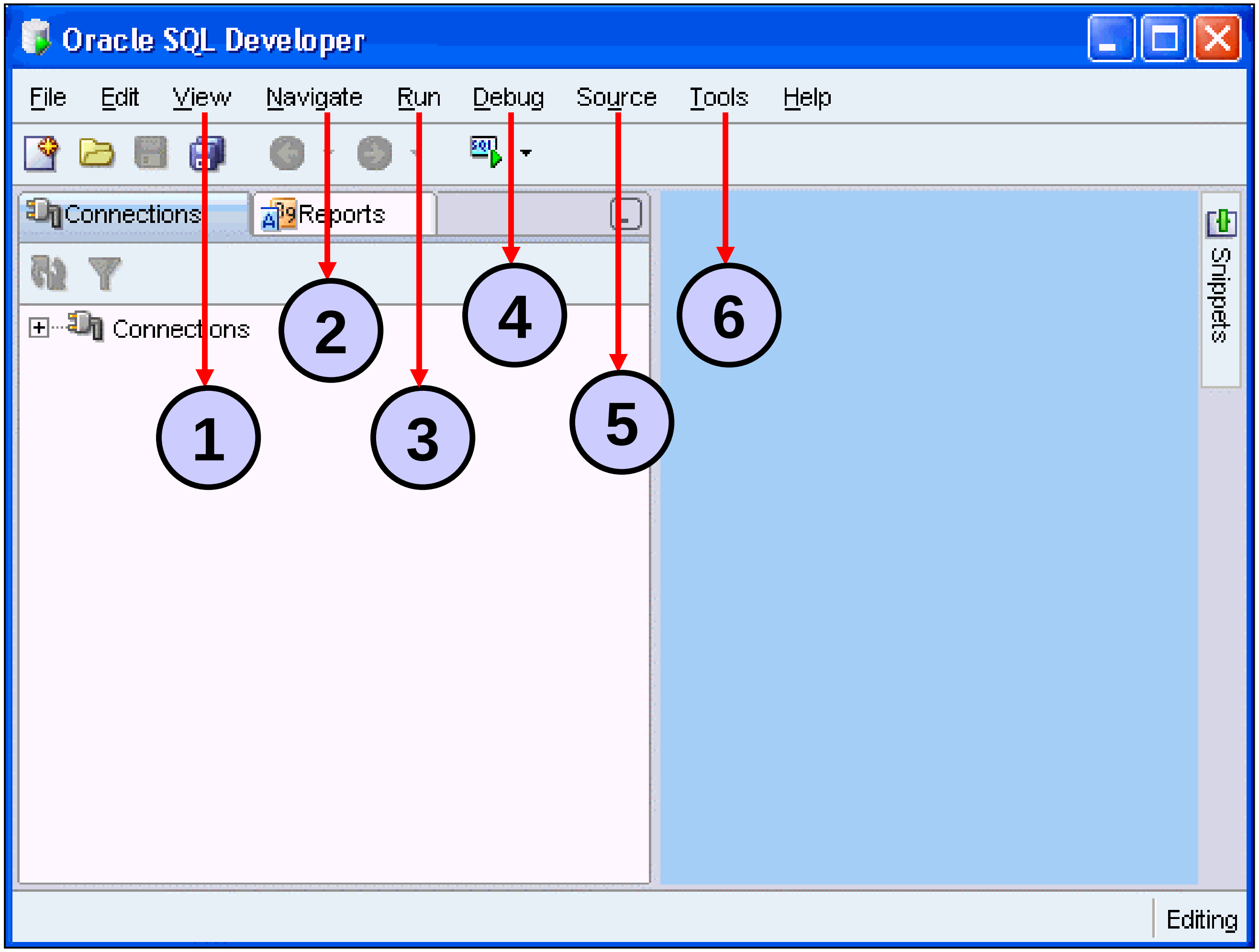
- Developed in Java
- Supports Windows, Linux, and Mac OS X platforms
- Default connectivity by using the JDBC Thin driver
- Does not require an installer
- Connects to any Oracle Database version 9.2.0.1 and later
- Bundled with JRE 1.5

Installing SQL Developer

Download the Oracle SQL Developer kit and unzip it into any directory on your machine.



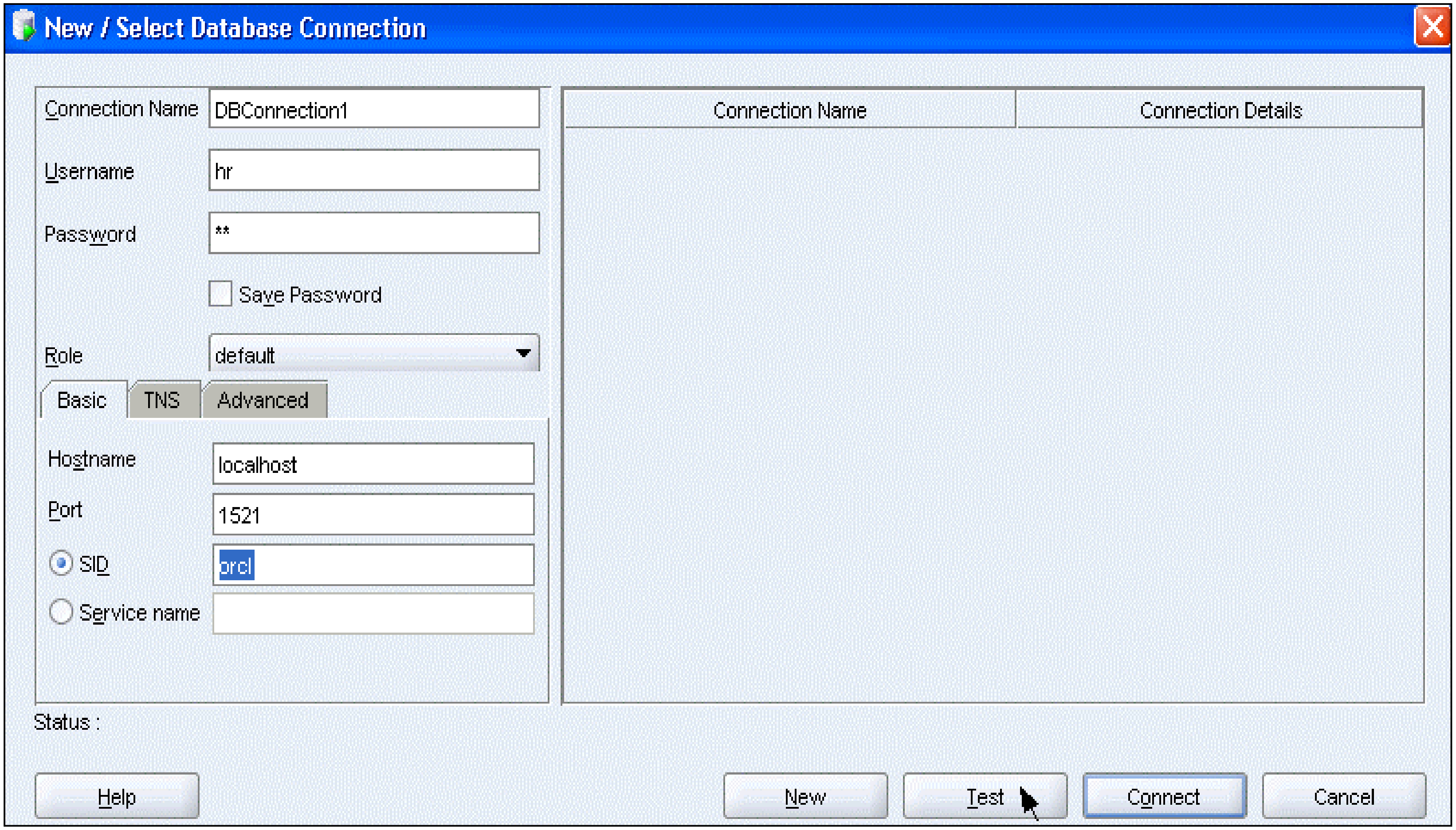
Menus for SQL Developer



Creating a Database Connection

- You must have at least one database connection to use SQL Developer.
- You can create and test connections:
  - For multiple databases
  - For multiple schemas
- SQL Developer automatically imports any connections defined in the `tnsnames.ora` file on your system.
- You can export connections to an XML file.
- Each additional database connection created is listed in the Connections Navigator hierarchy.

Creating a Database Connection

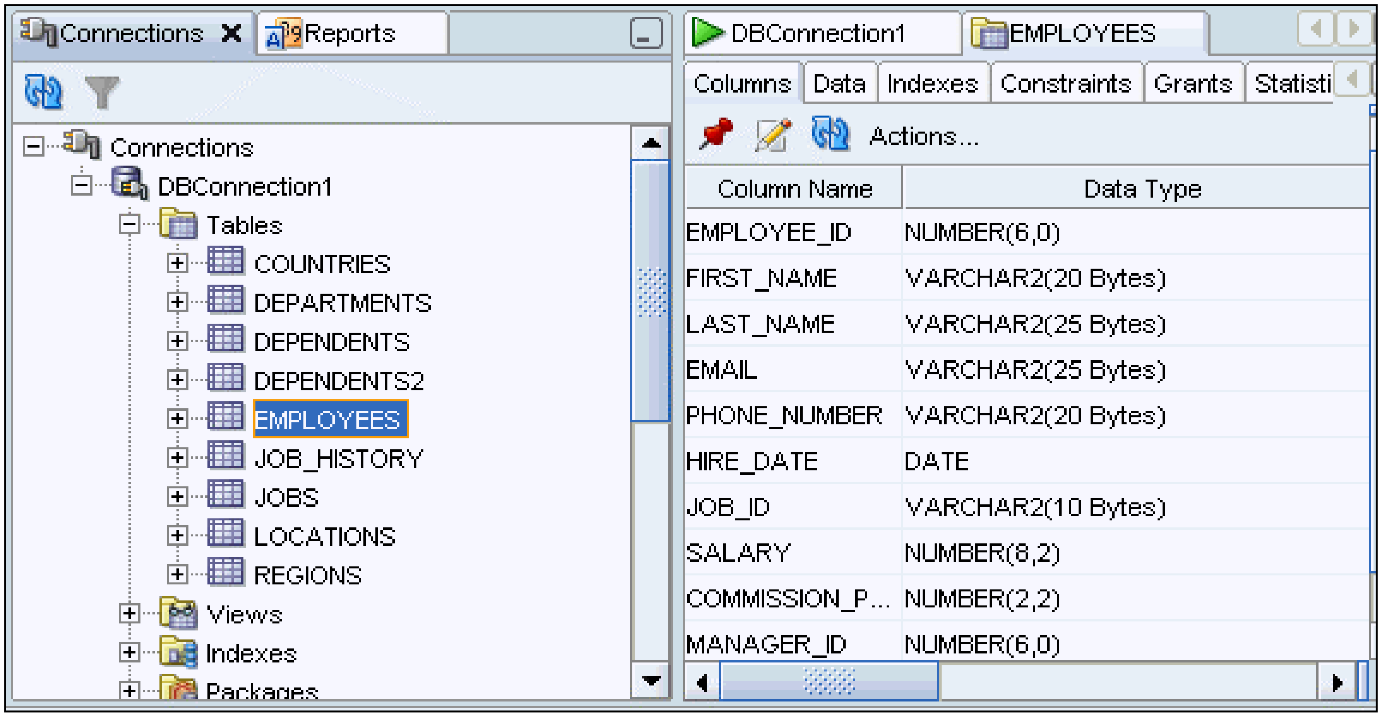


ORACLE

Browsing Database Objects

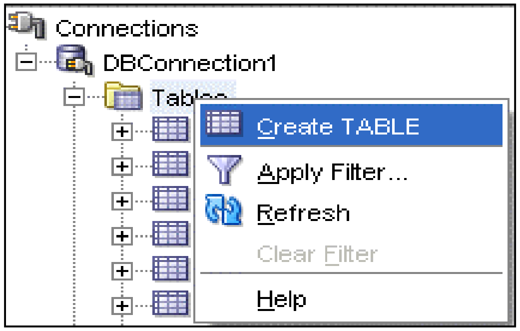
Use the Database Navigator to:

- **Browse through many objects in a database schema**
- **Review the definitions of objects at a glance**

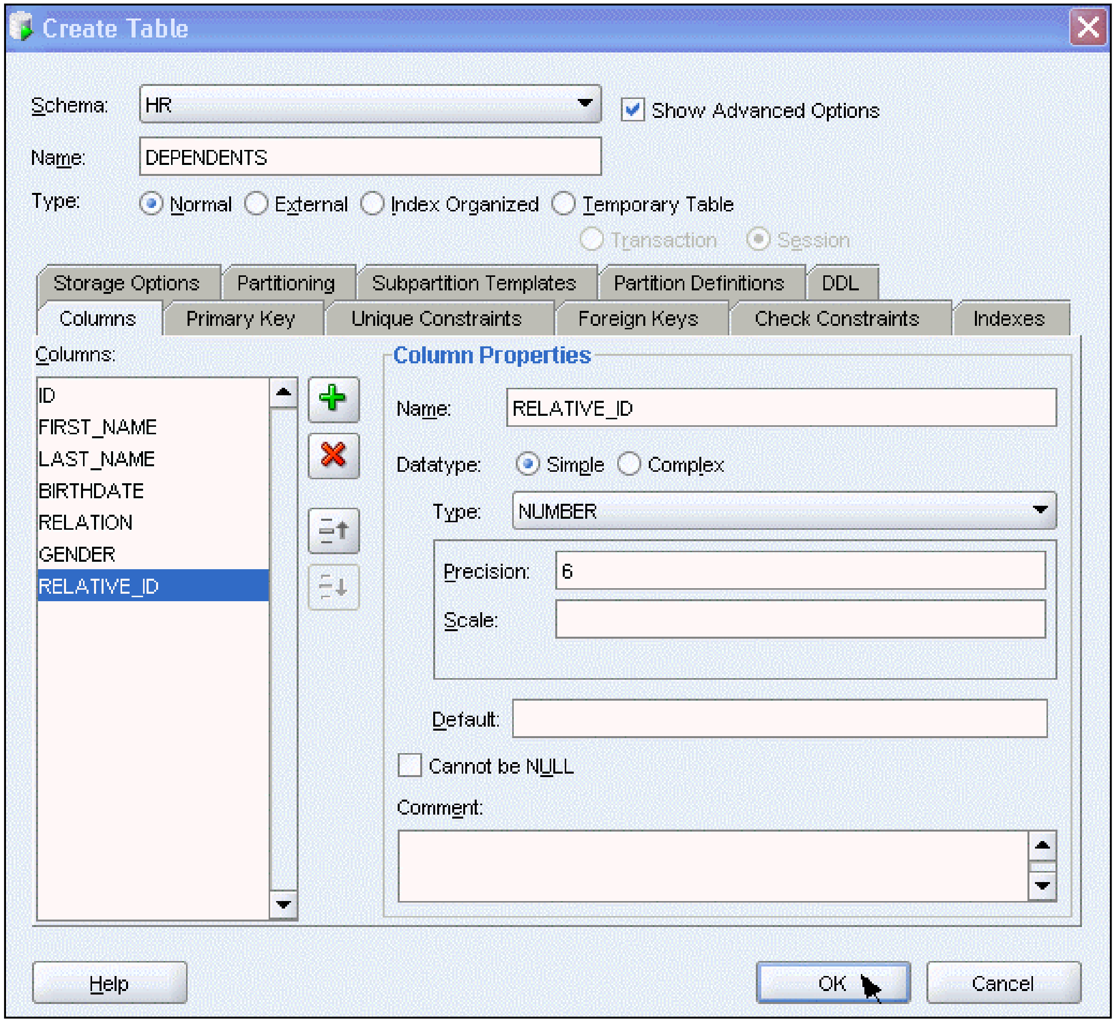


Creating a Schema Object

- SQL Developer supports the creation of any schema object by:
  - Executing a SQL statement in the SQL Worksheet
  - Using the context menu
- Edit the objects using an edit dialog box or one of the many context-sensitive menus.
- View the DDL for adjustments such as creating a new object or editing an existing schema object.



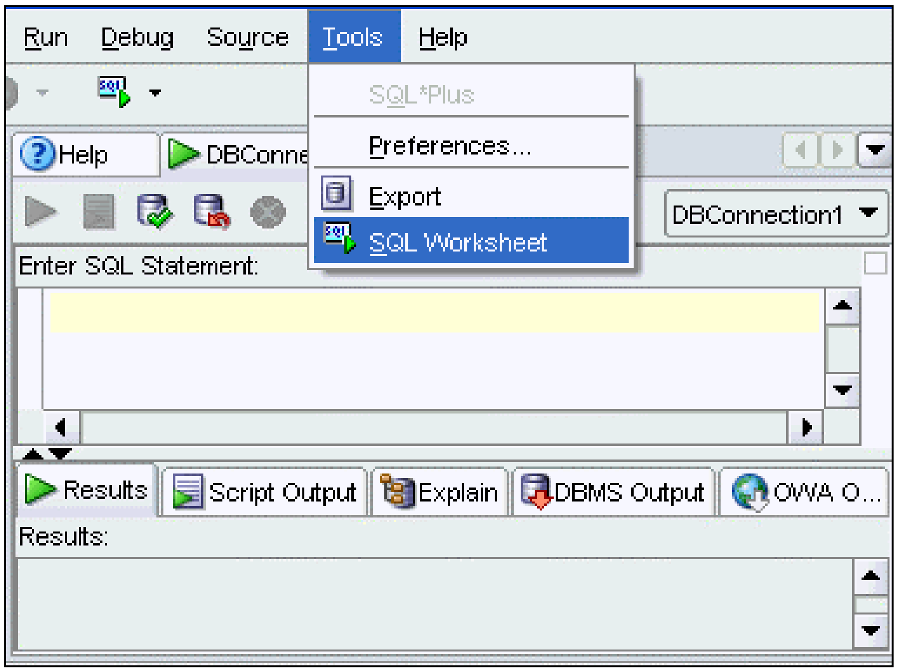
Creating a New Table: Example



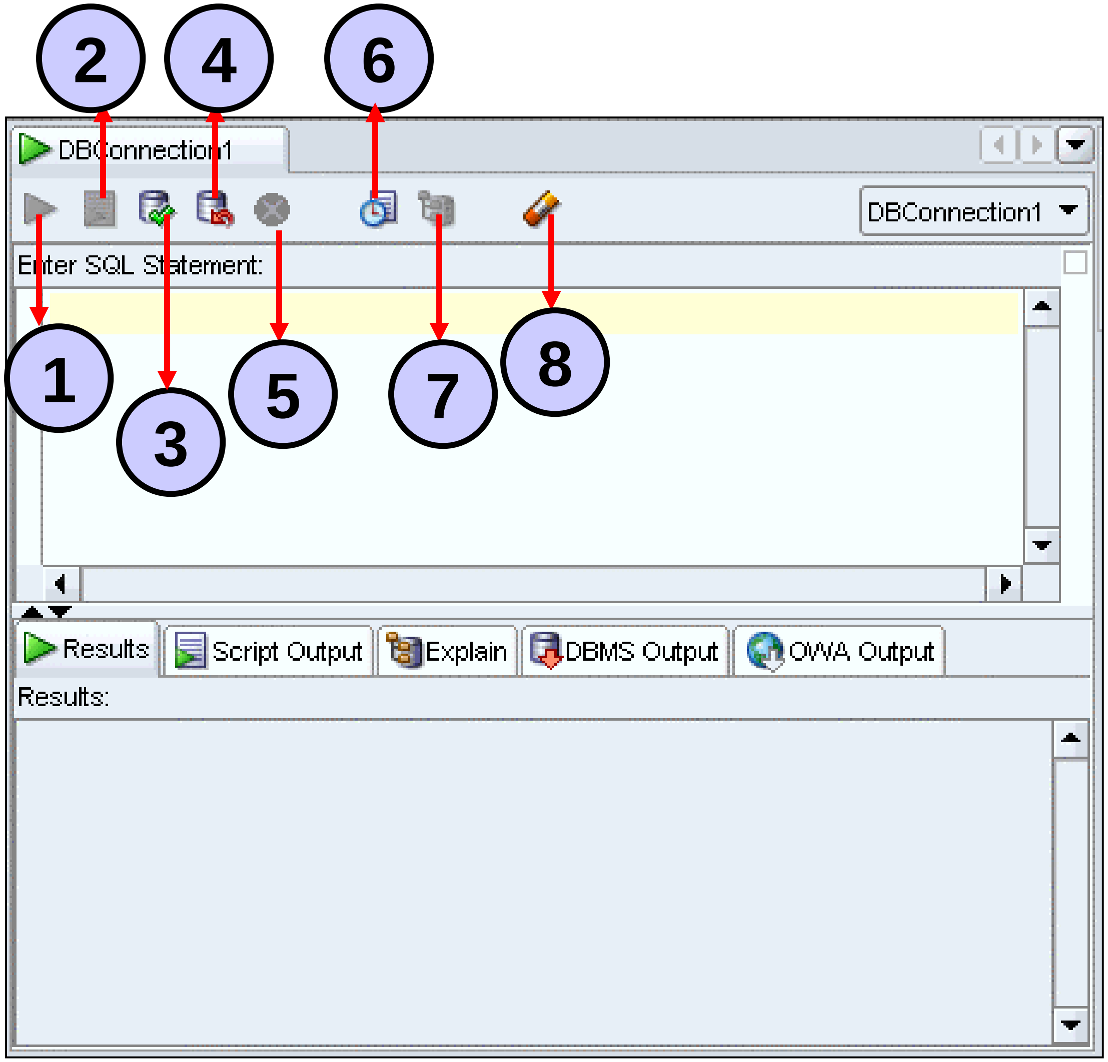
ORACLE

Using the SQL Worksheet

- Use the SQL Worksheet to enter and execute SQL, PL/SQL, and SQL *Plus statements.
- Specify any actions that can be processed by the database connection associated with the worksheet.

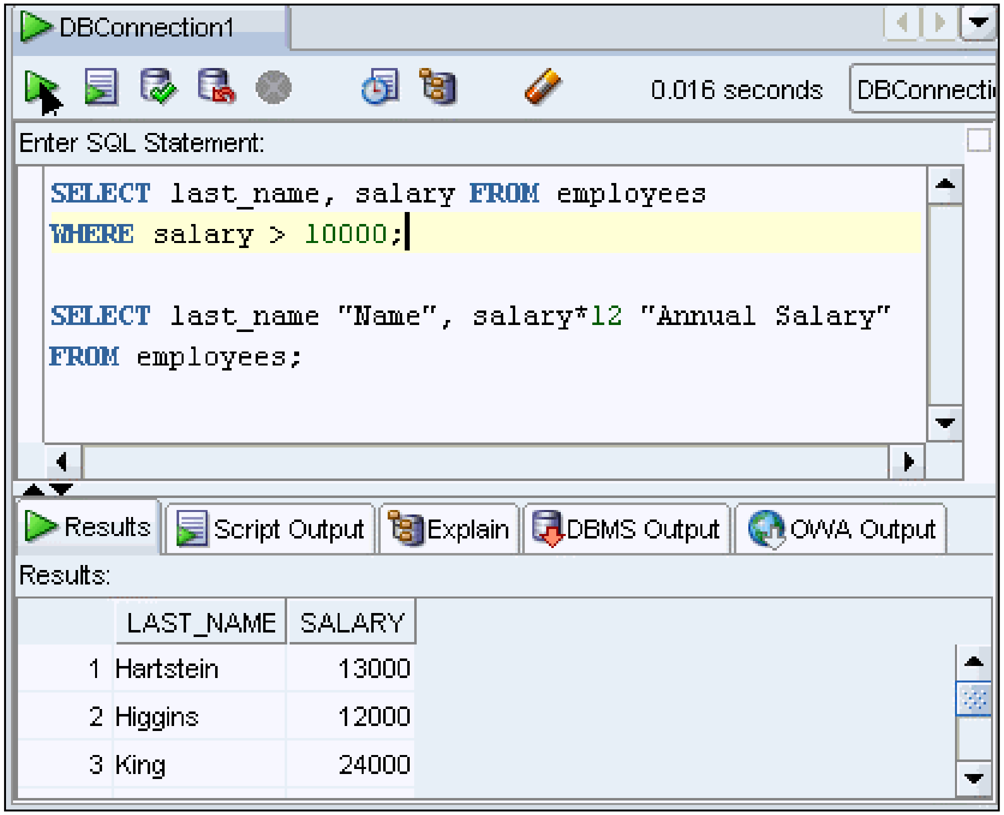


Using the SQL Worksheet



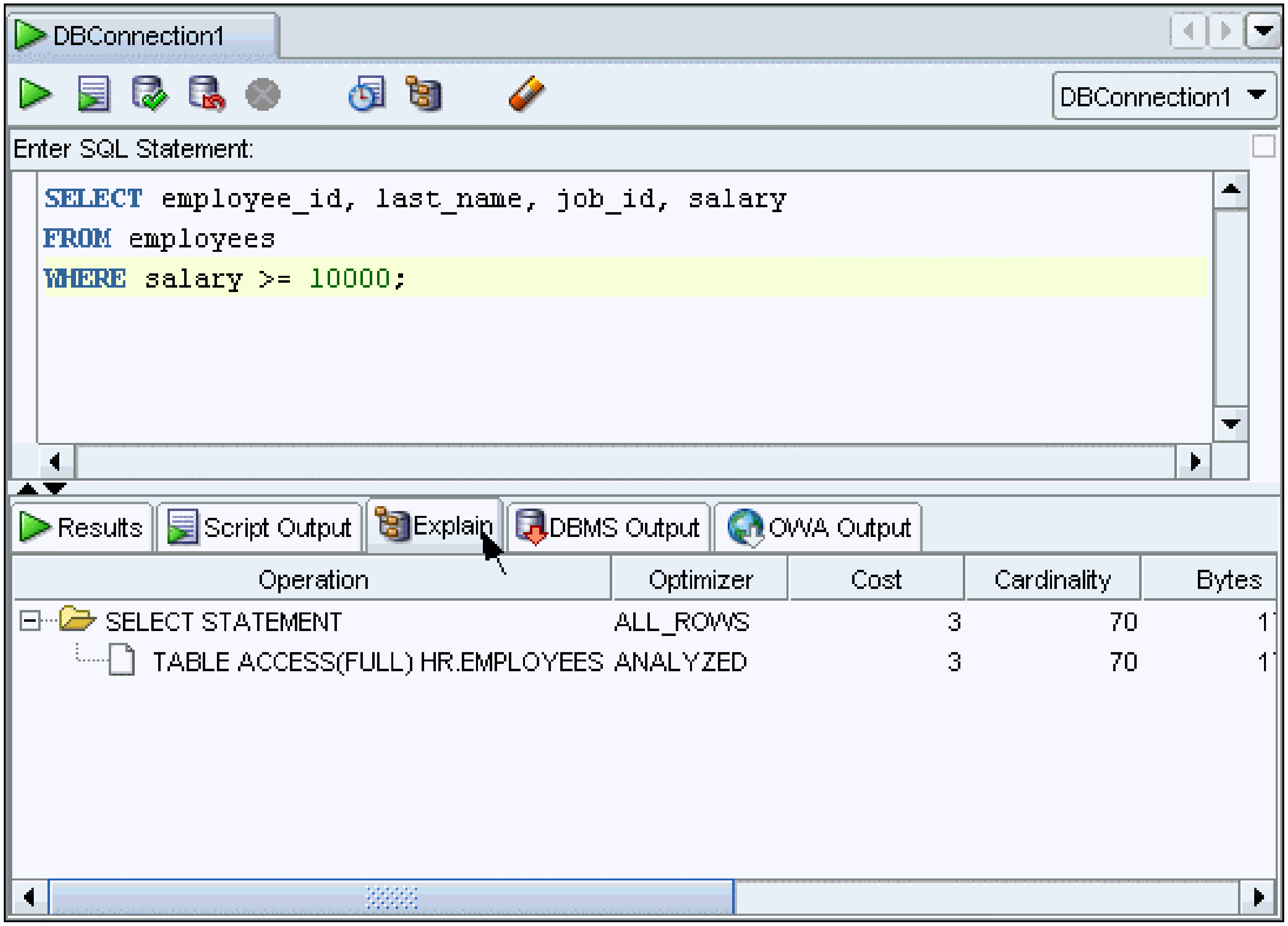
Executing SQL Statements

Use the Enter SQL Statement box to enter single or multiple SQL statements.



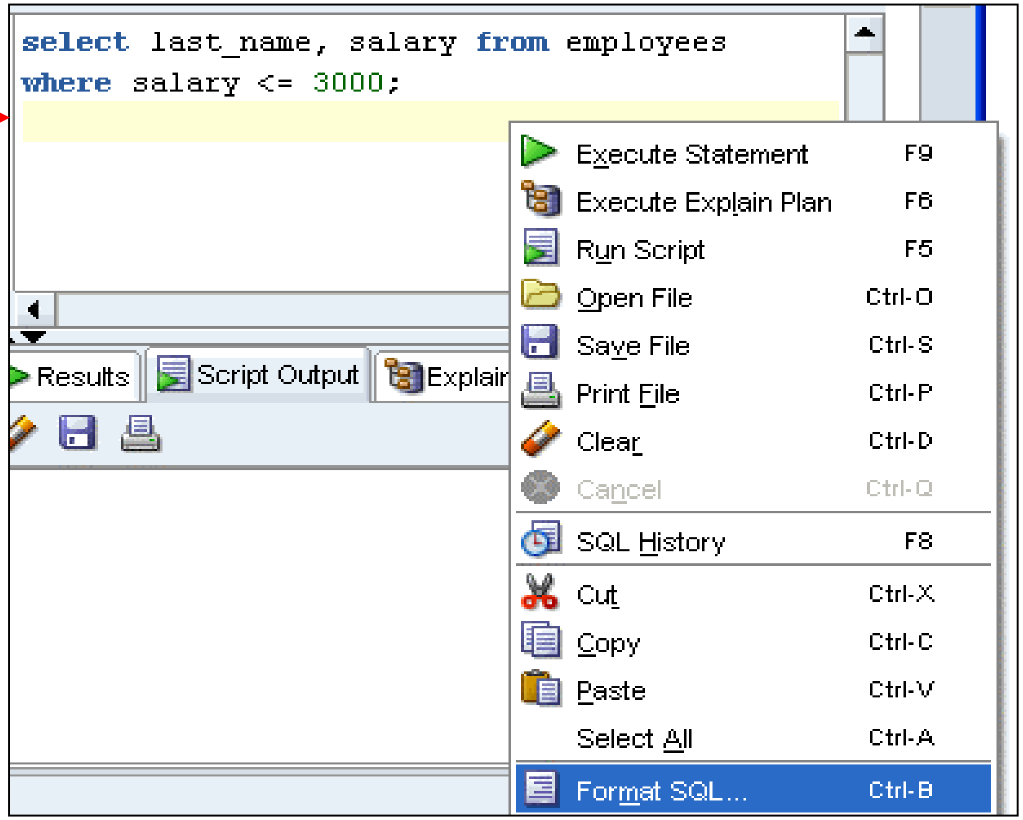
ORACLE

Viewing the Execution Plan

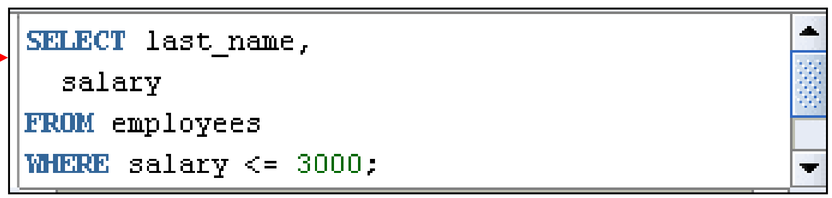


Formatting the SQL Code

Before formatting

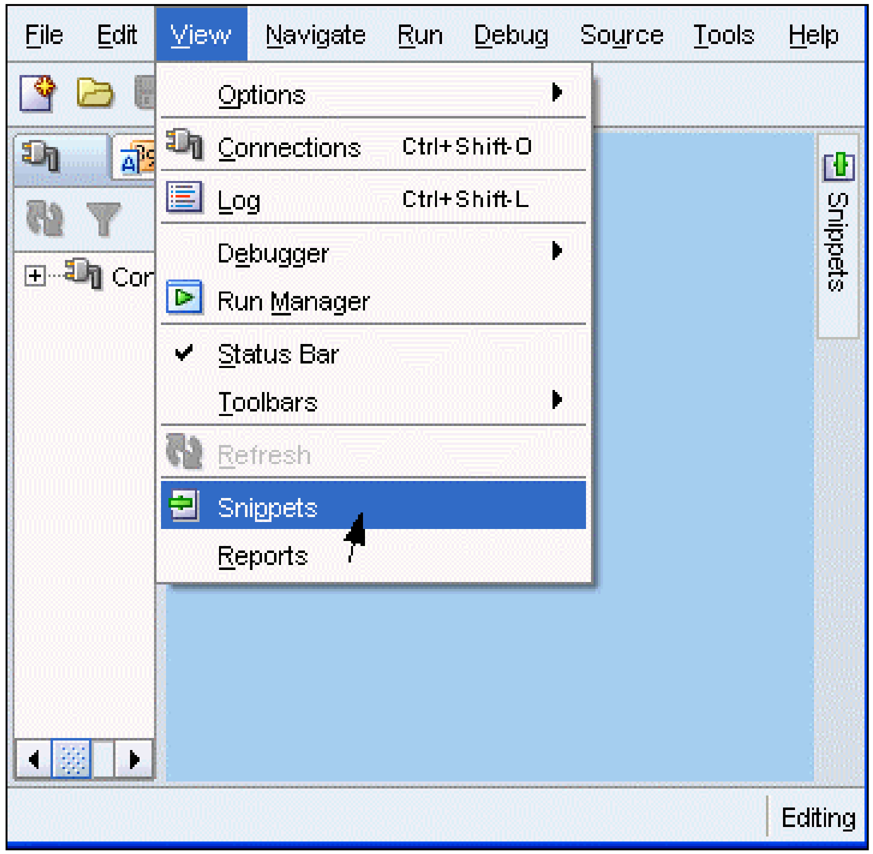


After formatting



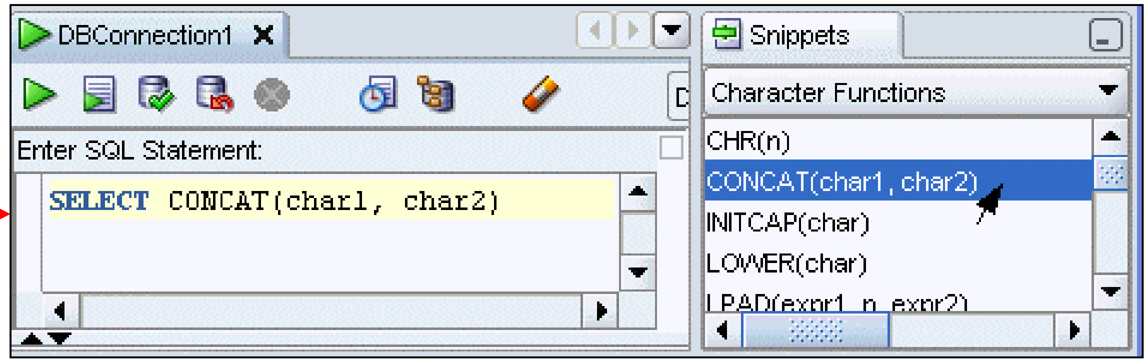
Using Snippets

Snippets are code fragments that may be just syntax or examples.

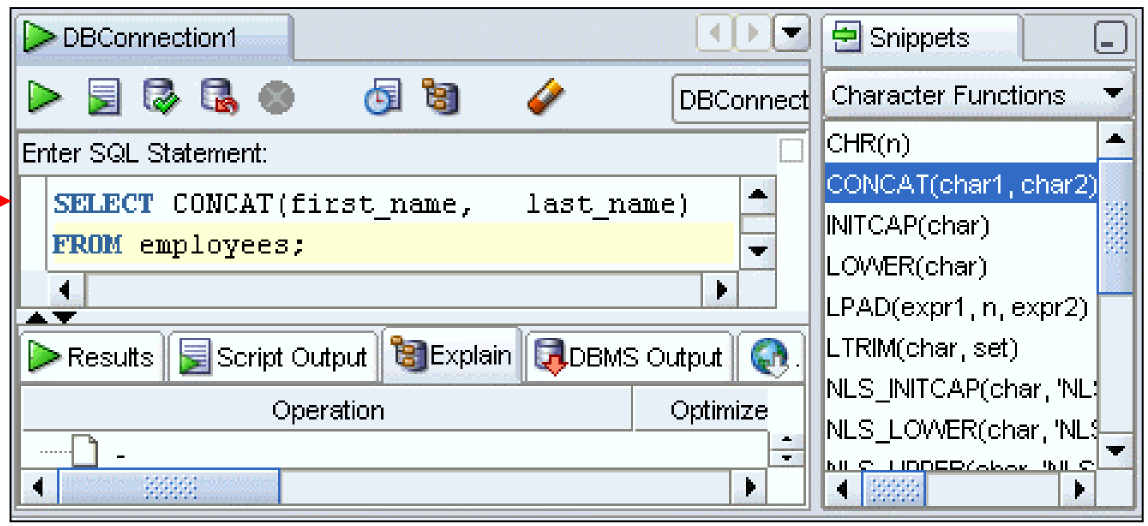


Using Snippets: Example

Inserting a snippet

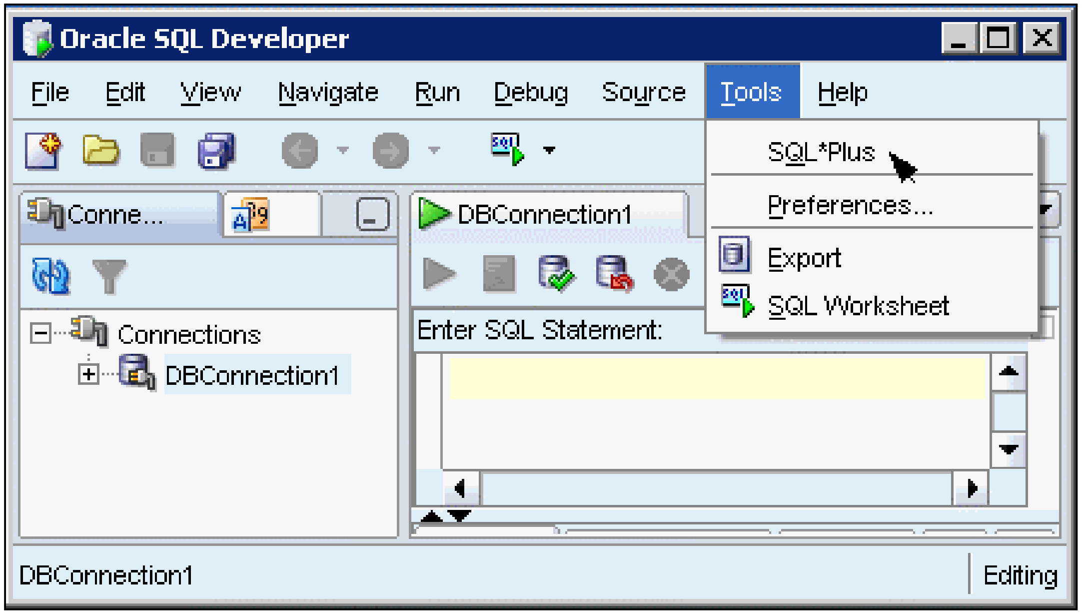


Editing the snippet



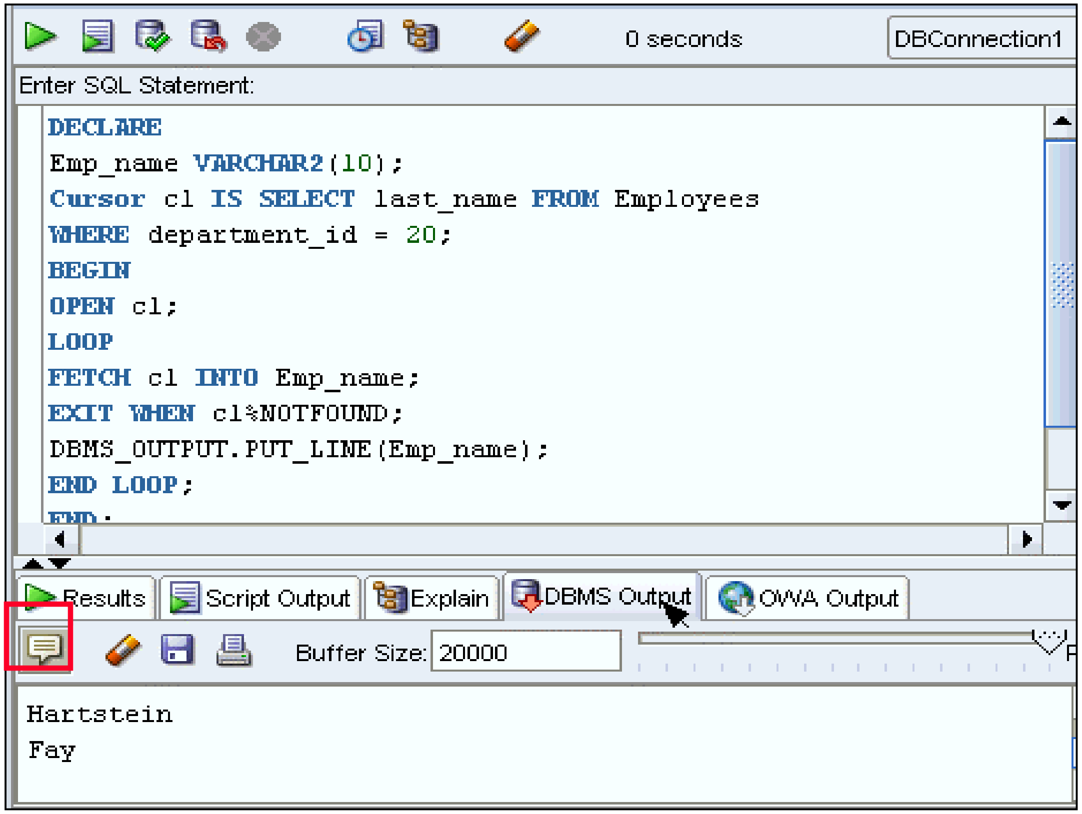
Using SQL*Plus

- The SQL Worksheet does not support all SQL*Plus statements.
- You can invoke the SQL*Plus command-line interface from SQL Developer.



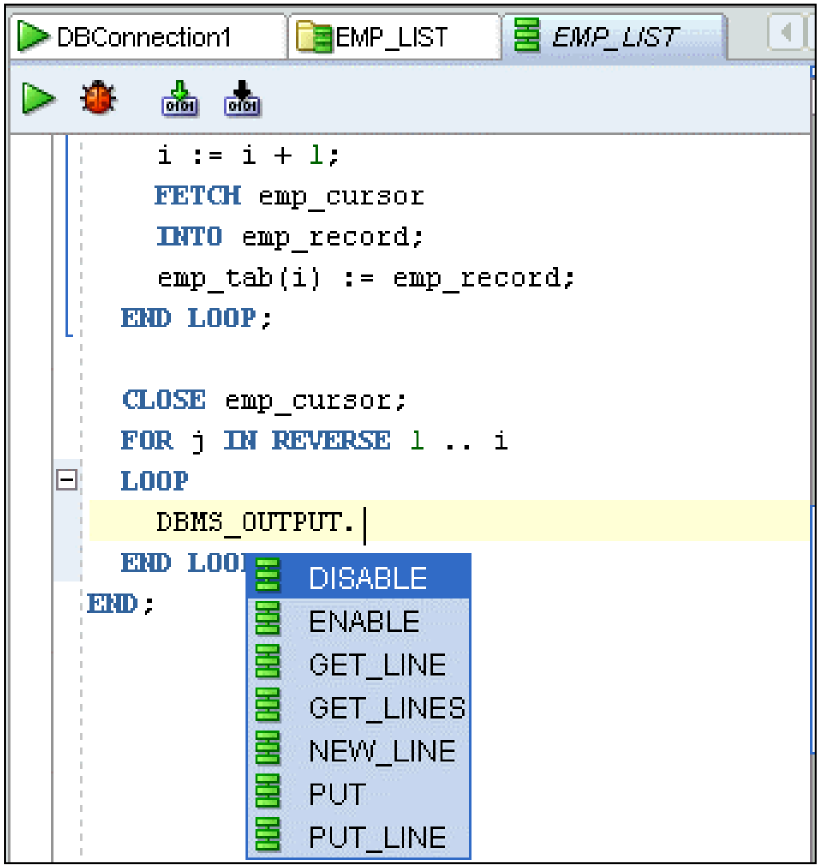
Creating an Anonymous Block

Create an anonymous block and display the output of the `DBMS_OUTPUT` package statements.

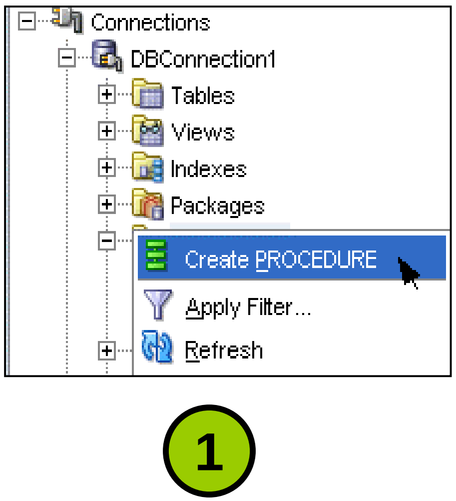


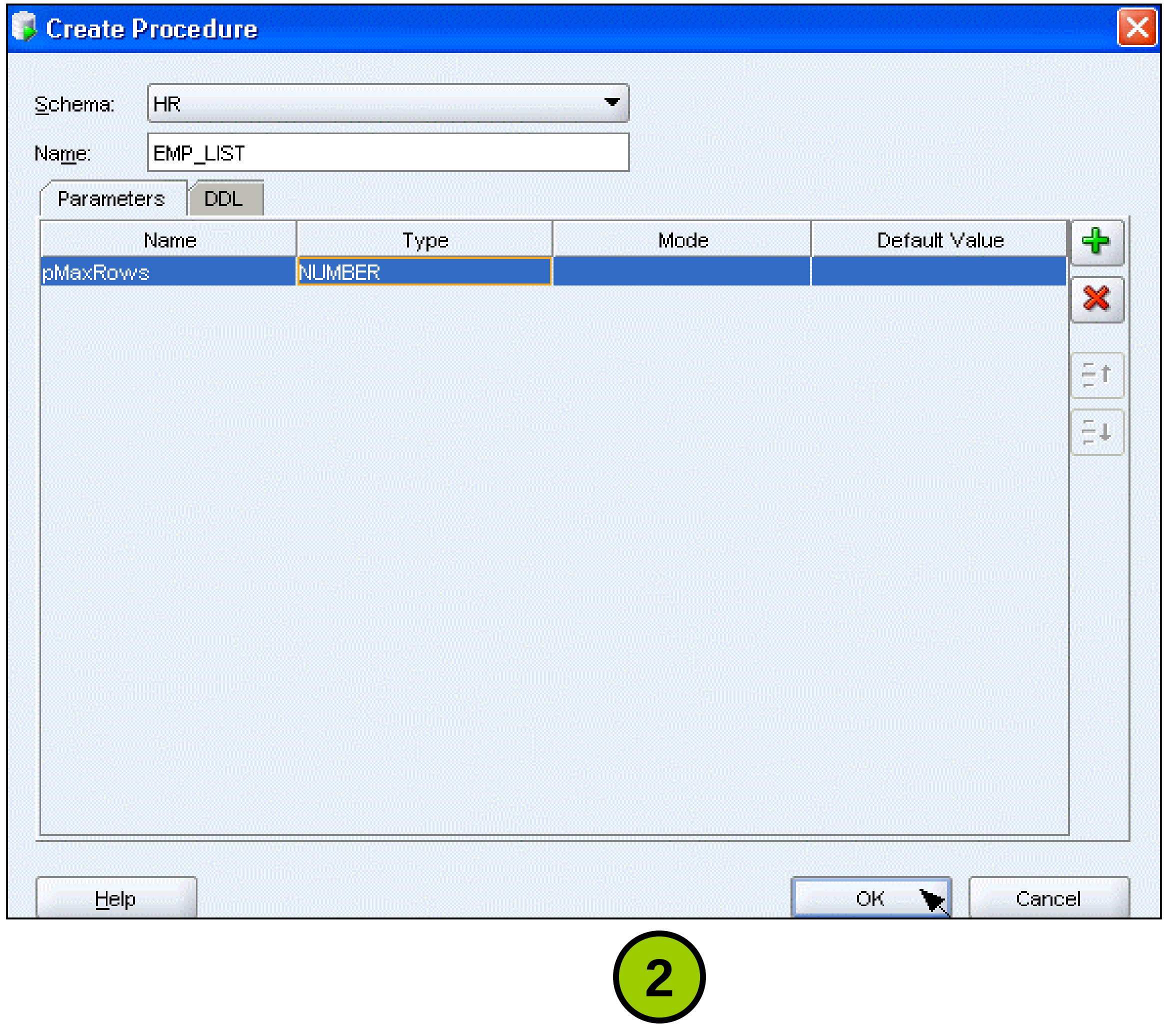
Editing the PL/SQL Code

Use the full-featured editor for PL/SQL program units.



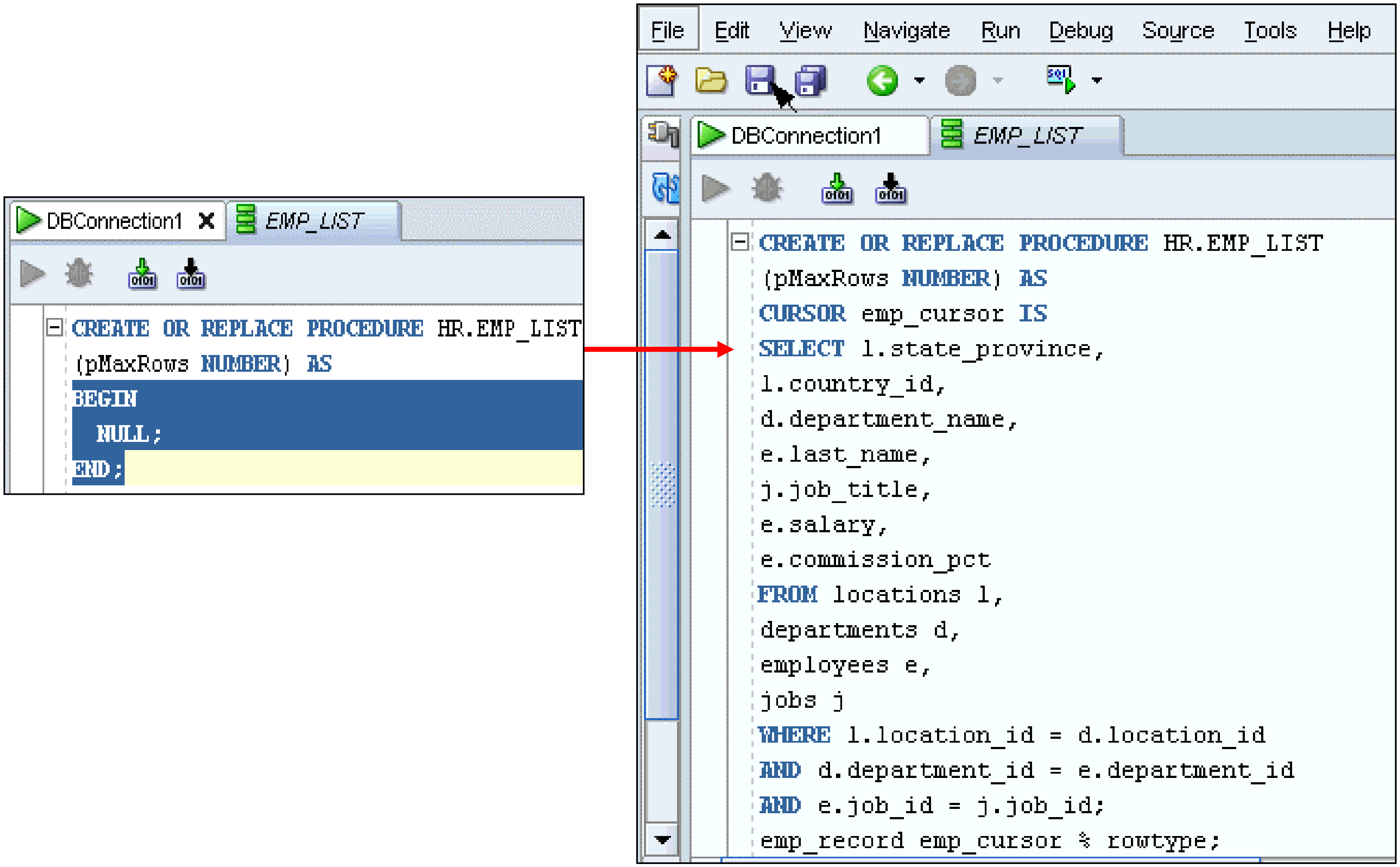
Creating a PL/SQL Procedure





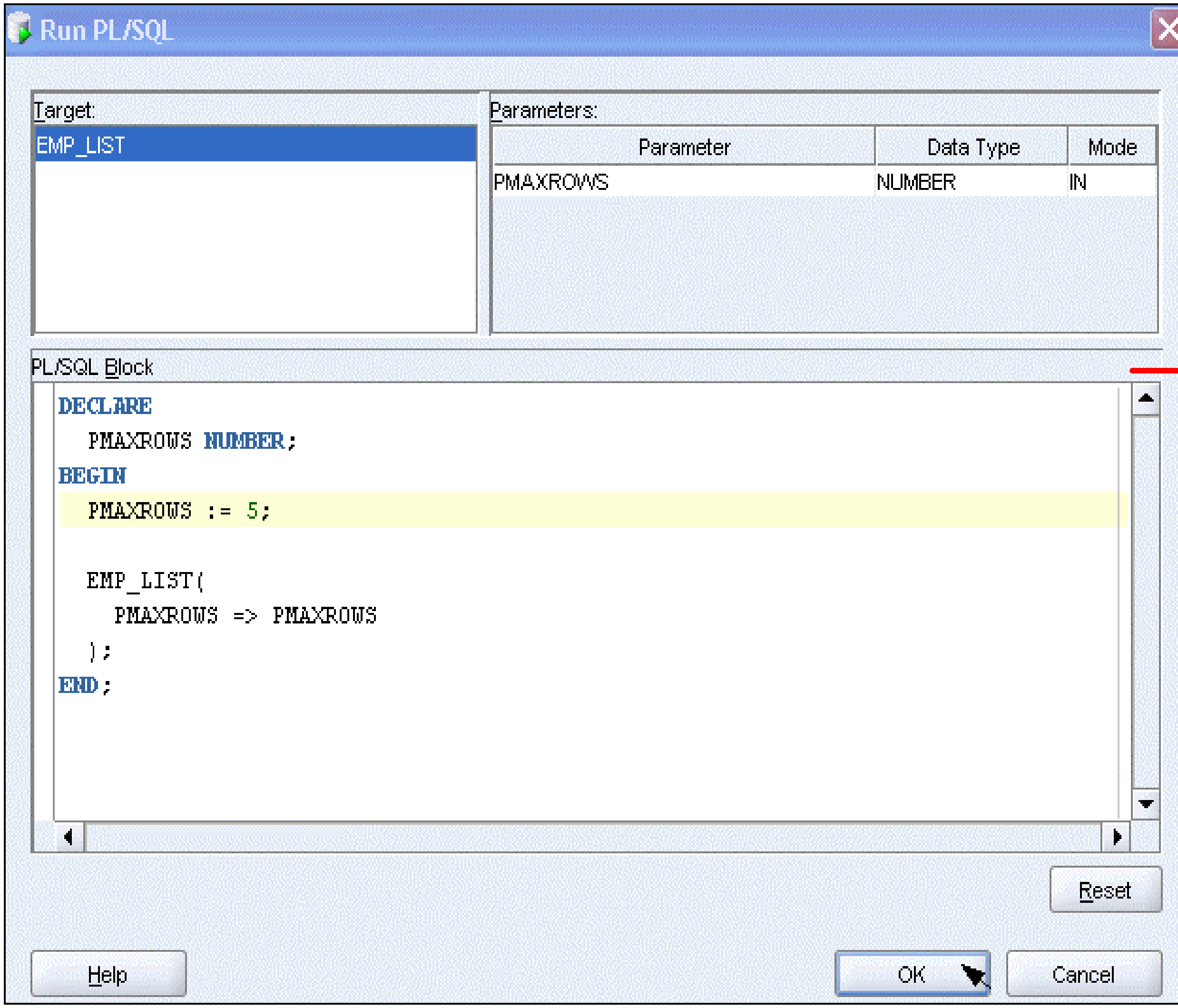
ORACLE

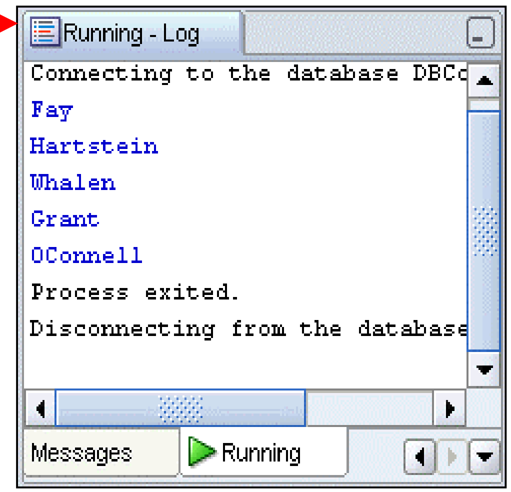
Compiling a PL/SQL Procedure



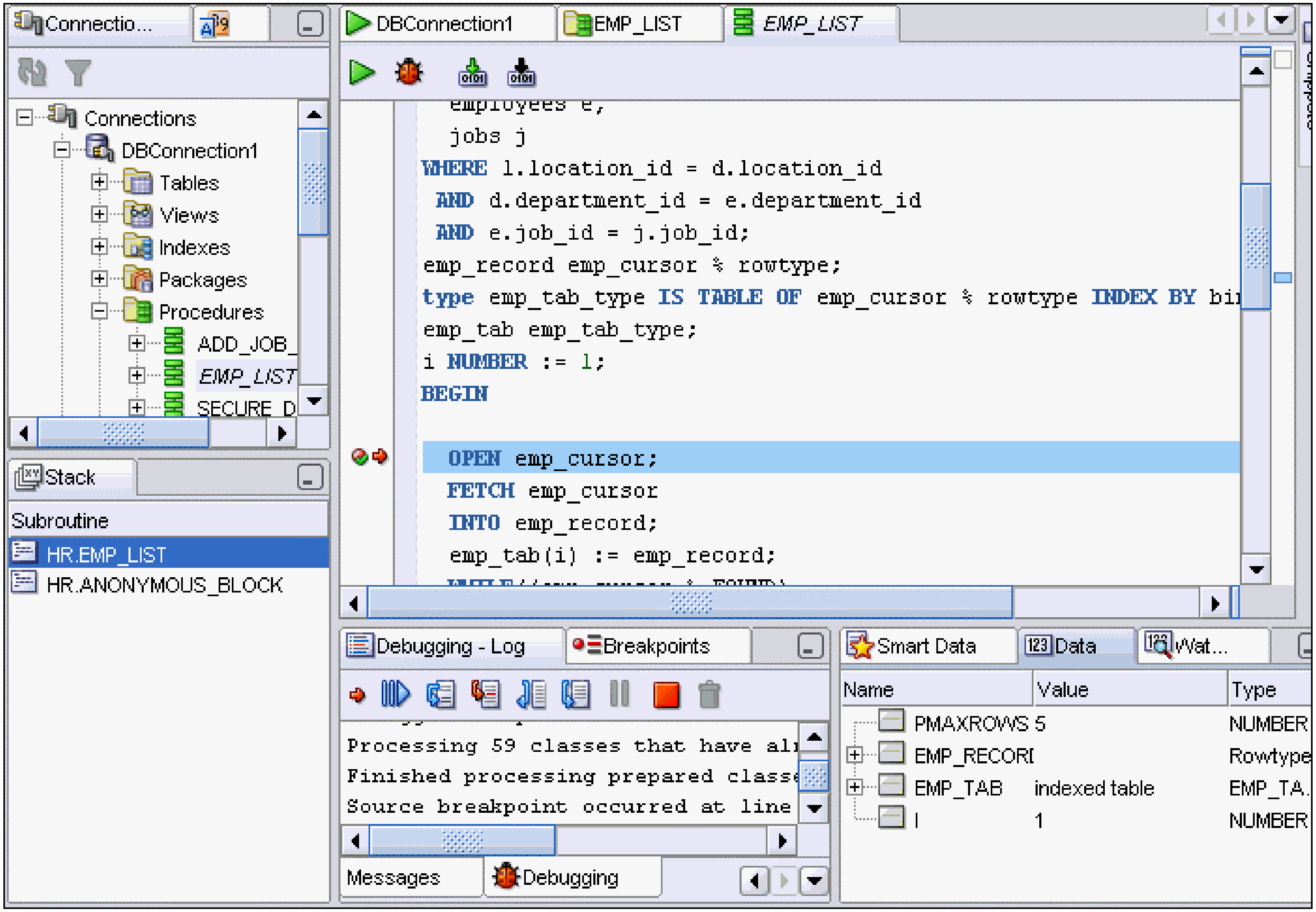
ORACLE

Running a PL/SQL Procedure



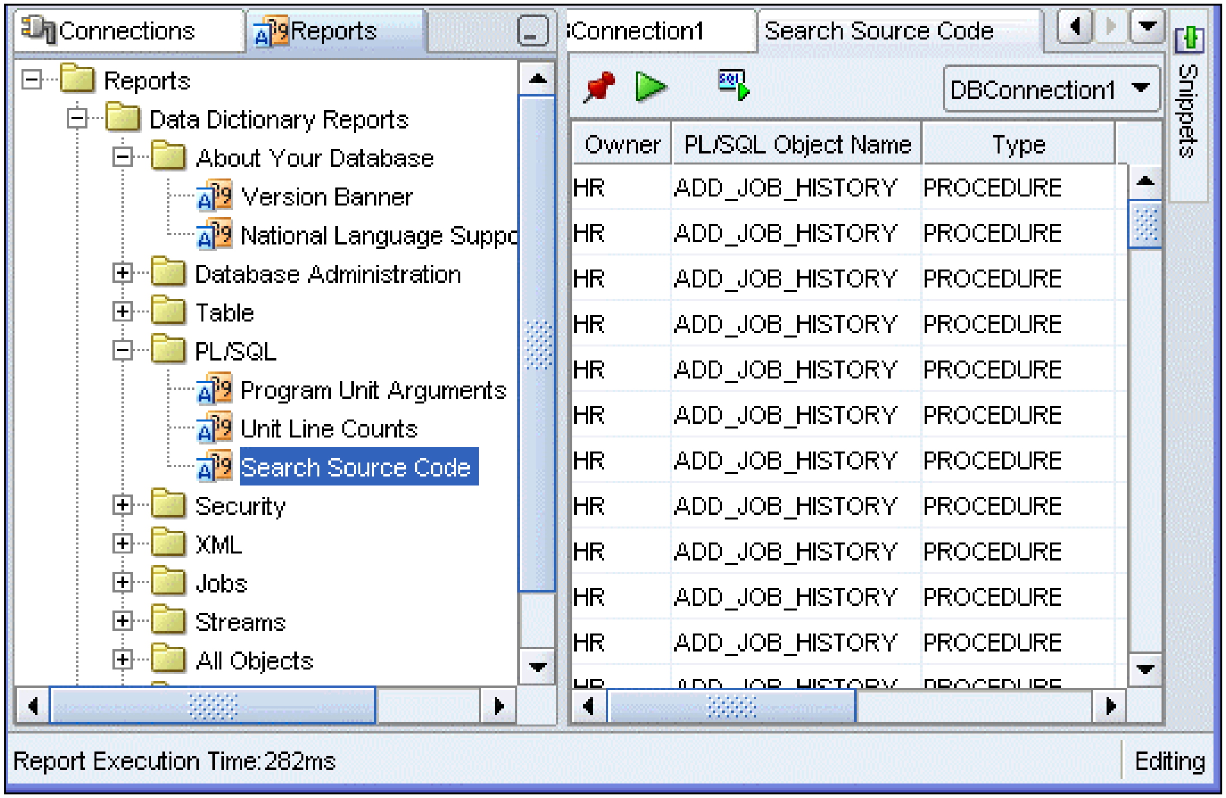


Debugging PL/SQL



Database Reporting

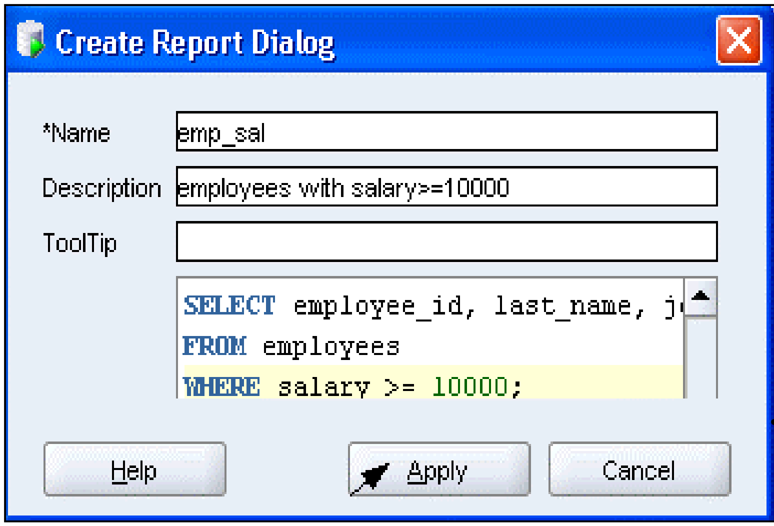
SQL Developer provides a number of predefined reports about the database and its objects.

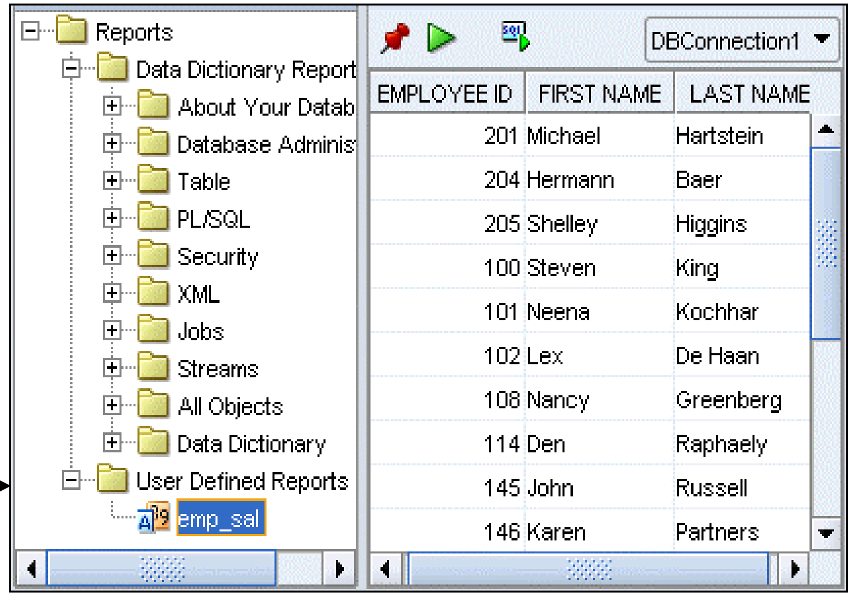


ORACLE

Creating a User-Defined Report

Create and save user-defined reports for repeated use.





Summary

In this appendix, you should have learned how to use SQL Developer to do the following:

- Browse, create, and edit database objects
- Execute SQL statements and scripts in the SQL Worksheet
- Edit and debug PL/SQL statements
- Create and save custom reports