

---

# Oracle Database 10g: PL/SQL Fundamentals

Electronic Presentation

---

D17112GC30  
Edition 3.0  
April 2009

**ORACLE®**

## **Authors**

Salome Clement  
Sunitha Patel  
Tulika Srivastava

## **Technical Contributors and Reviewers**

Brian Boxx  
Christoph Burandt  
Zarko Cesljas  
Dairy Chan  
Isabelle Cornu  
Kathryn Cunningham  
Laszlo Czinkoczki  
Burt Demchick  
Laura Garza  
Joel Goodman  
Nancy Greenberg  
Joe Greenwald  
Jonathan Grove  
Punita Handa  
Jessie Ho  
Craig Hollister  
Alison Holloway  
Chaitanya Koratamaddi  
Bryn Llewellyn  
Malika Marghadi  
Hildegard Mayr  
Miyuki Osato  
Nagavalli Pataballa  
Srinivas Putrevu  
Bryan Roberts

**Copyright © 2009, Oracle. All rights reserved.**

## **Disclaimer**

This document contains proprietary information and is protected by copyright and other intellectual property laws. You may copy and print this document solely for your own use in an Oracle training course. The document may not be modified or altered in any way. Except where your use constitutes "fair use" under copyright law, you may not use, share, download, upload, copy, print, display, perform, reproduce, publish, license, post, transmit, or distribute this document in whole or in part without the express authorization of Oracle.

The information contained in this document is subject to change without notice. If you find any problems in the document, please report them in writing to: Oracle University, 500 Oracle Parkway, Redwood Shores, California 94065 USA. This document is not warranted to be error-free.

## **Restricted Rights Notice**

If this documentation is delivered to the United States Government or anyone using the documentation on behalf of the United States Government, the following notice is applicable:

### **U.S. GOVERNMENT RIGHTS**

The U.S. Government's rights to use, modify, reproduce, release, perform, display, or disclose these training materials are restricted by the terms of the applicable Oracle license agreement and/or the applicable U.S. Government contract.

## **Trademark Notice**

Oracle is a registered trademark of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.

## **Technical Contributors and Reviewers**

Helen Robertson  
Grant Spencer  
Michael Versaci  
Lex Van Der Werff  
Yash Jain

## **Editors**

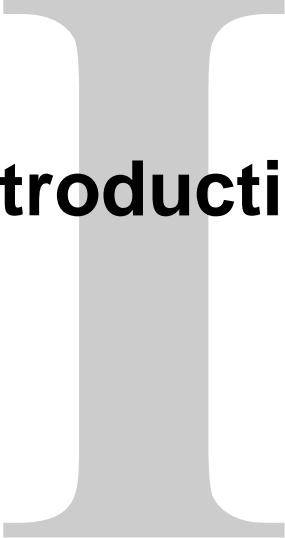
Richard Wallis  
Arijit Ghosh

## **Graphic Designers**

Steve Elwood  
Priya Saxena

## **Publishers**

Sujatha Nagendra  
Michael Sebastian  
Nita Brozowski  
Srividya Rameshkumar

A large, light gray, serif capital letter 'I' is centered on the page. The word 'Introduction' is written in a bold, black, sans-serif font across the middle of the 'I'.

# **Introduction**

# Objectives

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

- Describe the objectives of the course
- Describe the course agenda
- Identify the database tables used in the course
- Identify the Oracle products that help you design a complete business solution

# Course Objectives

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

- Describe how PL/SQL provides programming extensions to SQL
- Write PL/SQL code to interface with the database
- Design PL/SQL program units that execute efficiently
- Use PL/SQL programming constructs and conditional control statements
- Handle run-time errors
- Describe stored procedures and functions

# Course Agenda

Lessons for the first day:

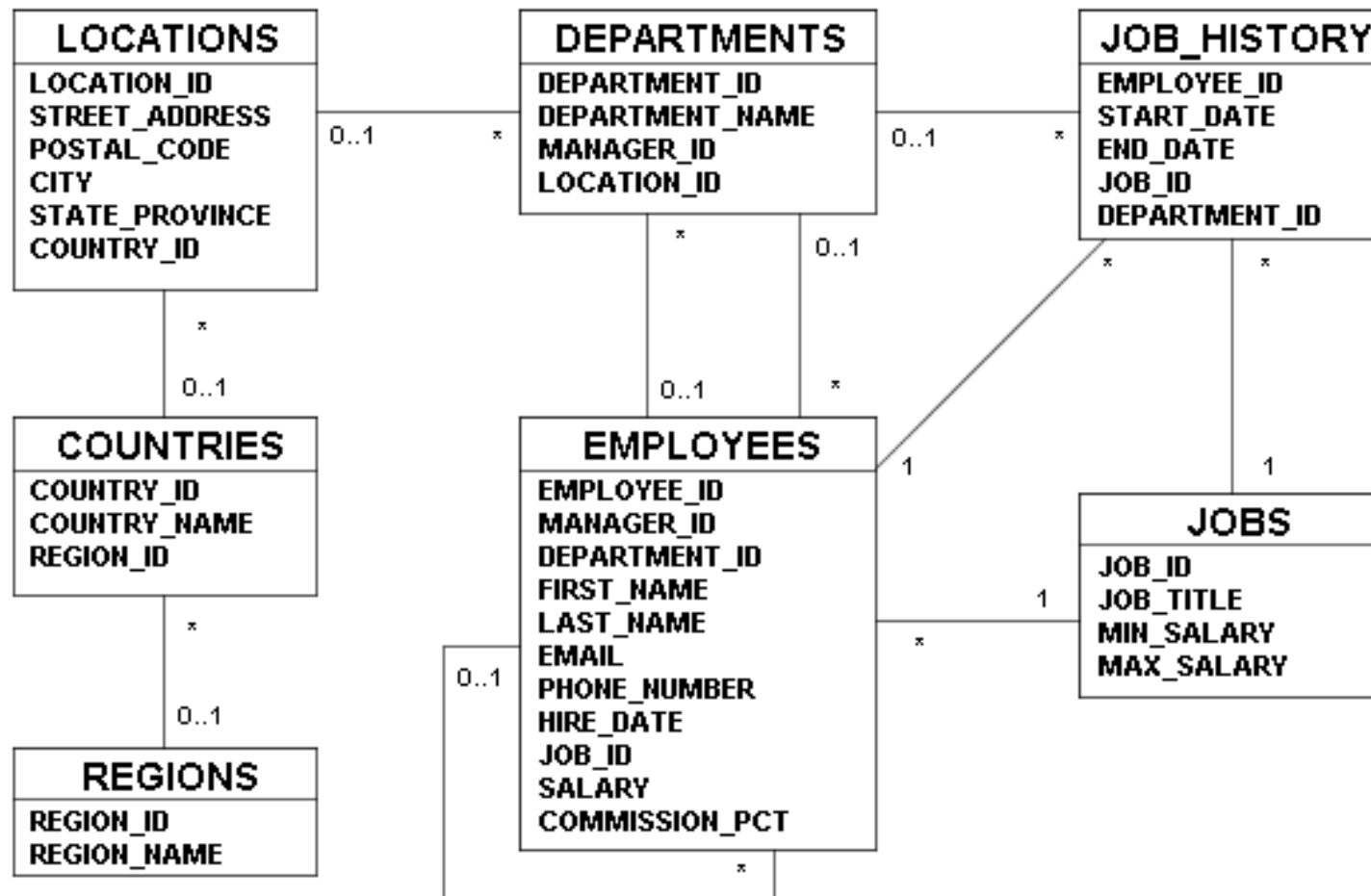
- I. Introduction
  - 1. Introduction to PL/SQL
  - 2. Declaring PL/SQL Variables
  - 3. Writing Executable Statements
  - 4. Interacting with the Oracle Server
  - 5. Writing Control Structures

# Course Agenda

Lessons for the second day:

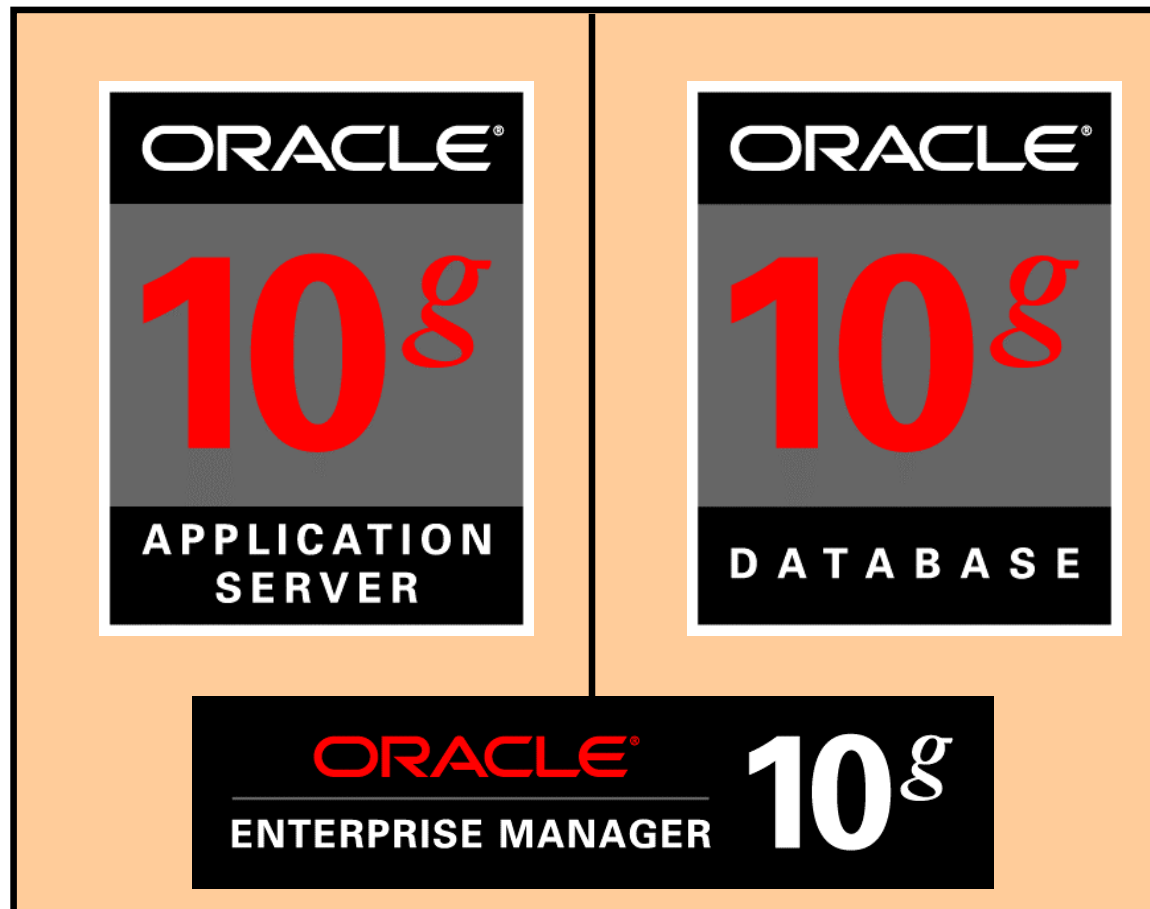
6. Working with Composite Data Types
7. Using Explicit Cursors
8. Handling Exceptions
9. Creating Stored Procedures and Functions

# Human Resources (hr) Data Set

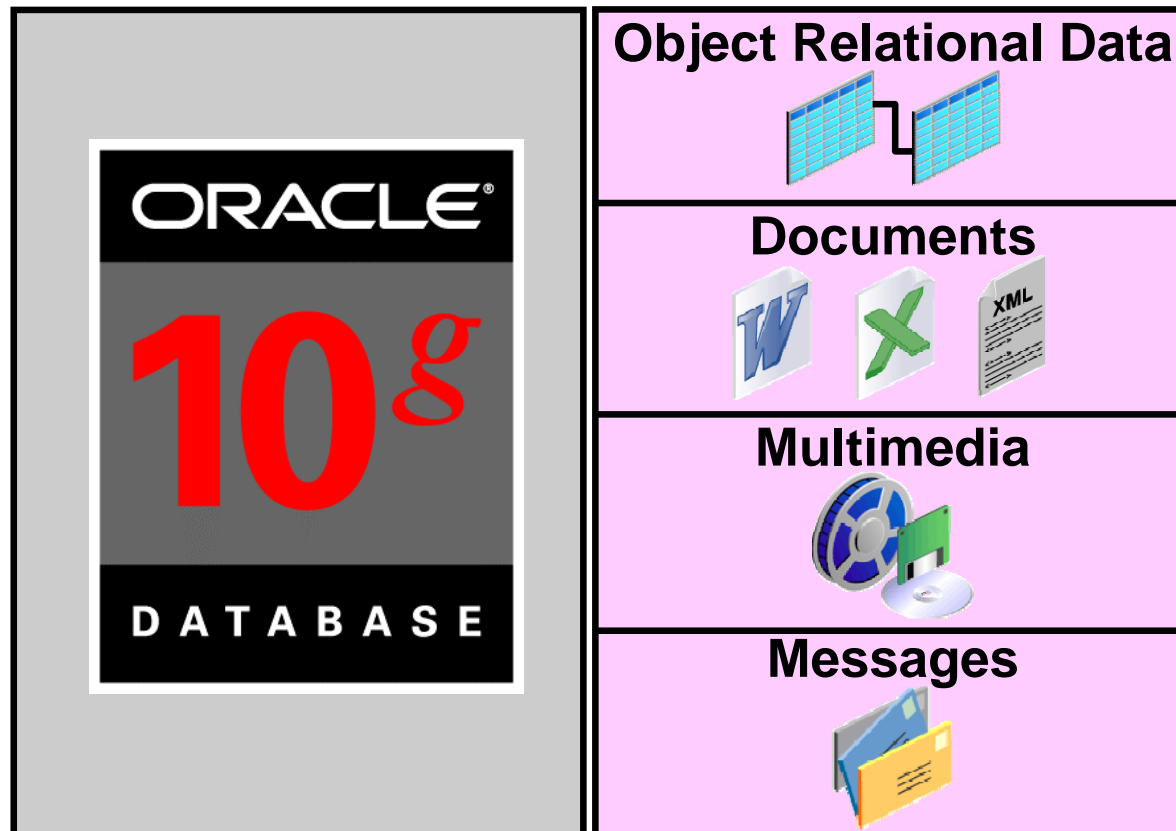




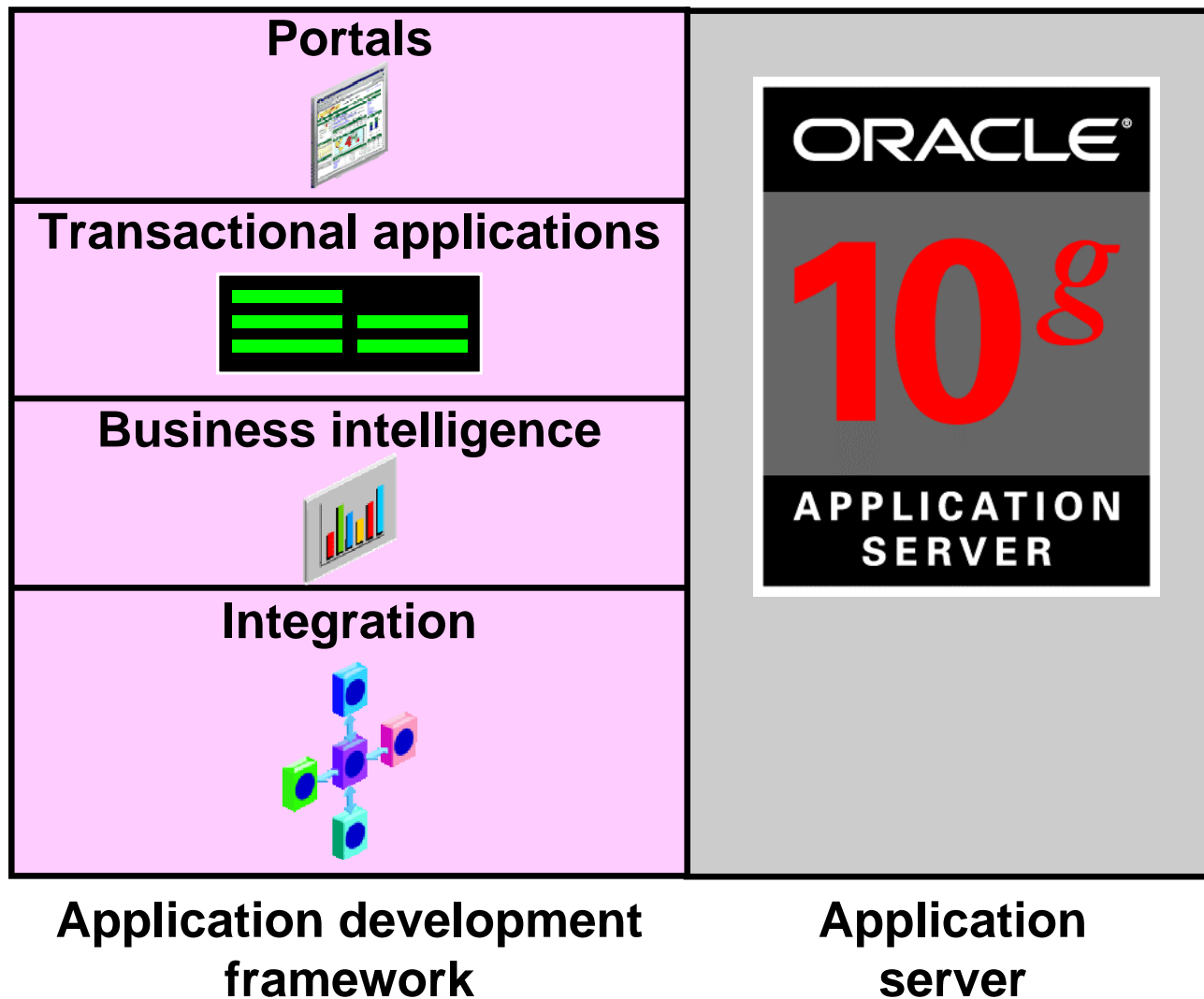
# Oracle10g Grid Infrastructure



# Oracle Database 10g



# Oracle Application Server 10g

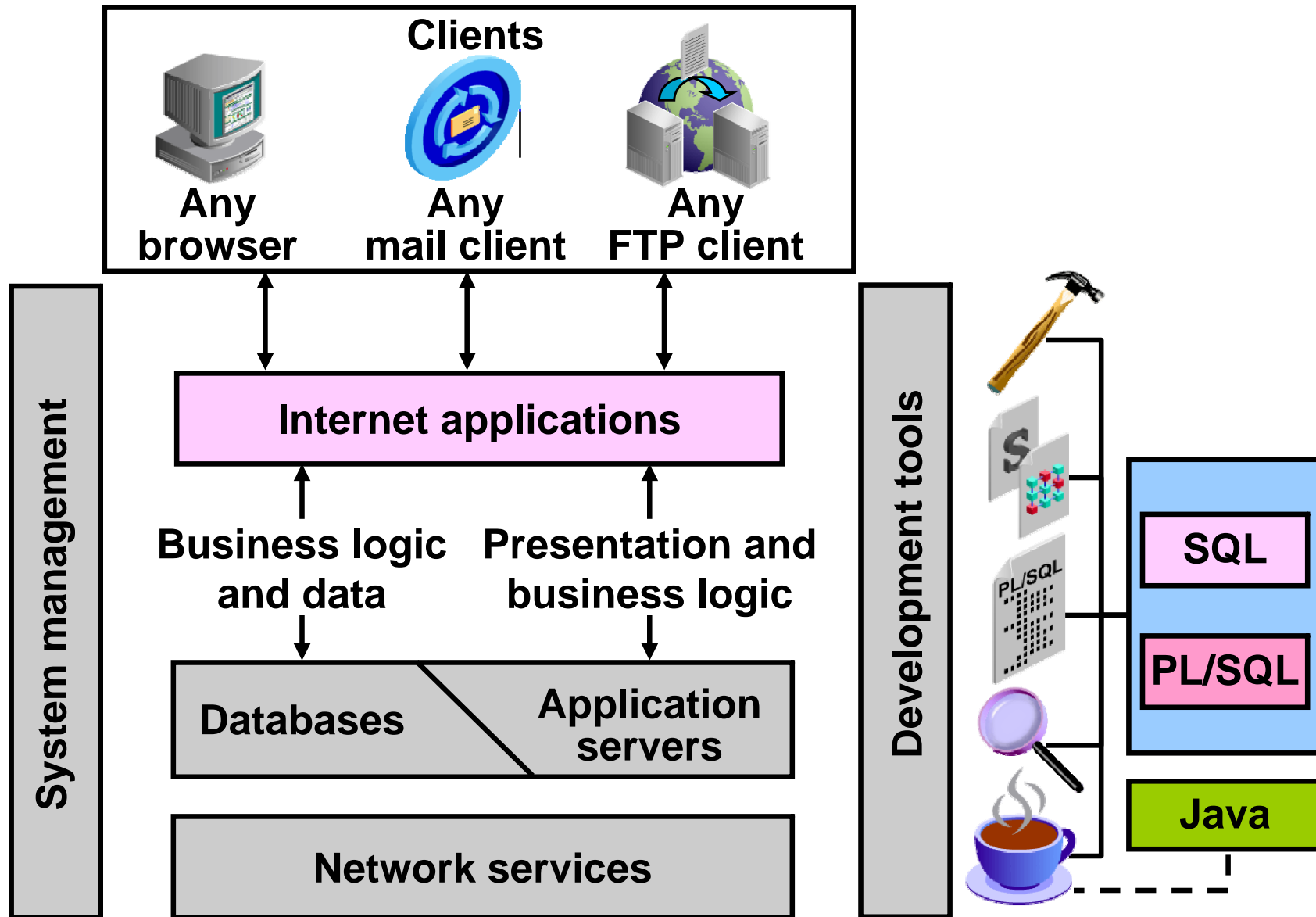


# Oracle Enterprise Manager 10g Grid Control

- Software provisioning
- Application service-level monitoring



# Oracle Internet Platform



# Summary

In this lesson, you should have learned how to:

- Describe the course objectives and course agenda
- Identify tables and their relationships in the `hr` schema
- Identify the various products in the Oracle 10g grid infrastructure that enable you to develop a complete business solution

# 1

## Introduction to PL/SQL

# Objectives

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

- Explain the need for PL/SQL
- Explain the benefits of PL/SQL
- Identify the different types of PL/SQL blocks
- Use Oracle SQL Developer as a development environment for PL/SQL
- Output messages from PL/SQL



# What Is PL/SQL?

## PL/SQL:

- Stands for Procedural Language extension to SQL
- Is Oracle Corporation's standard data access language for relational databases
- Seamlessly integrates procedural constructs with SQL

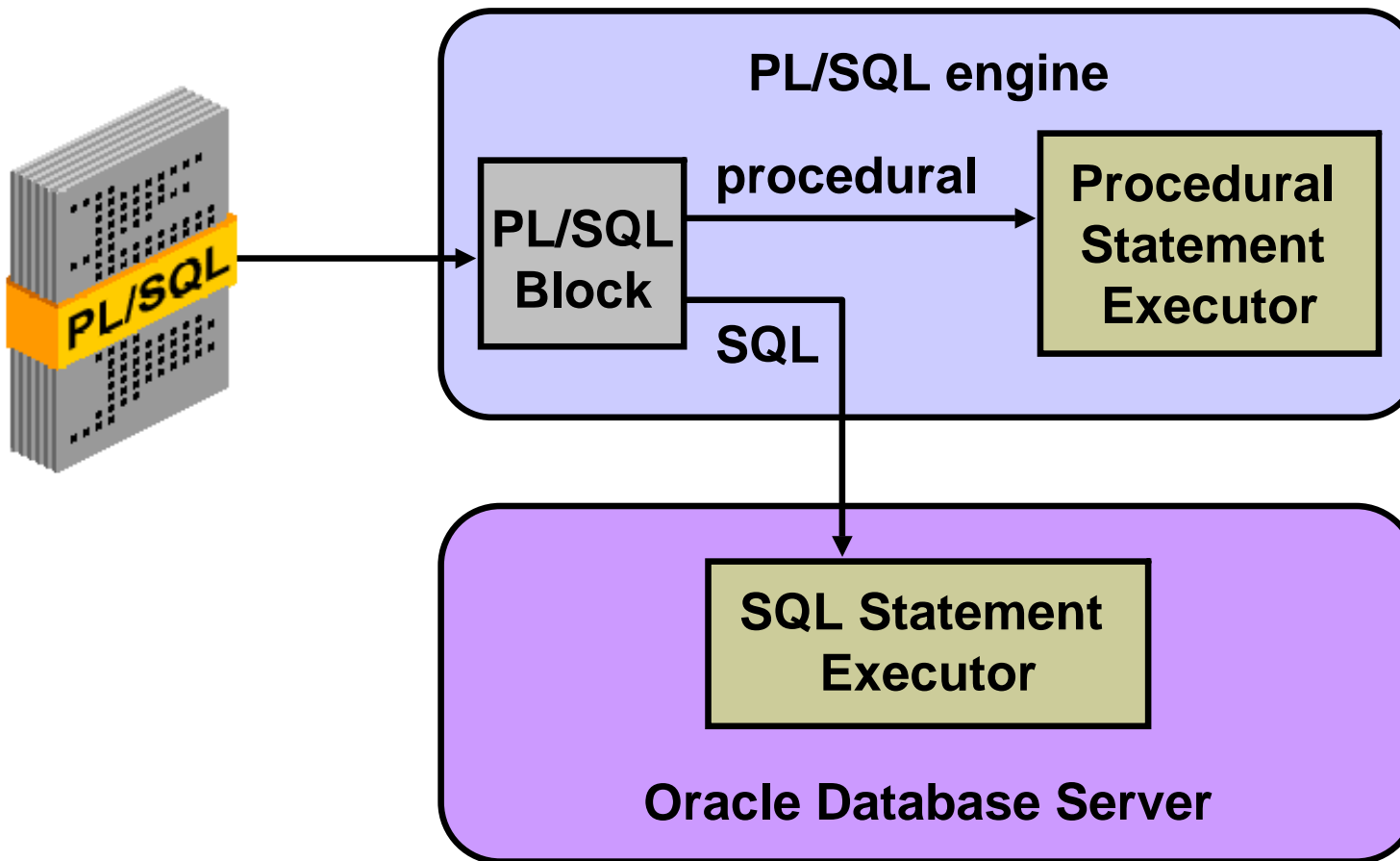


# About PL/SQL

## PL/SQL:

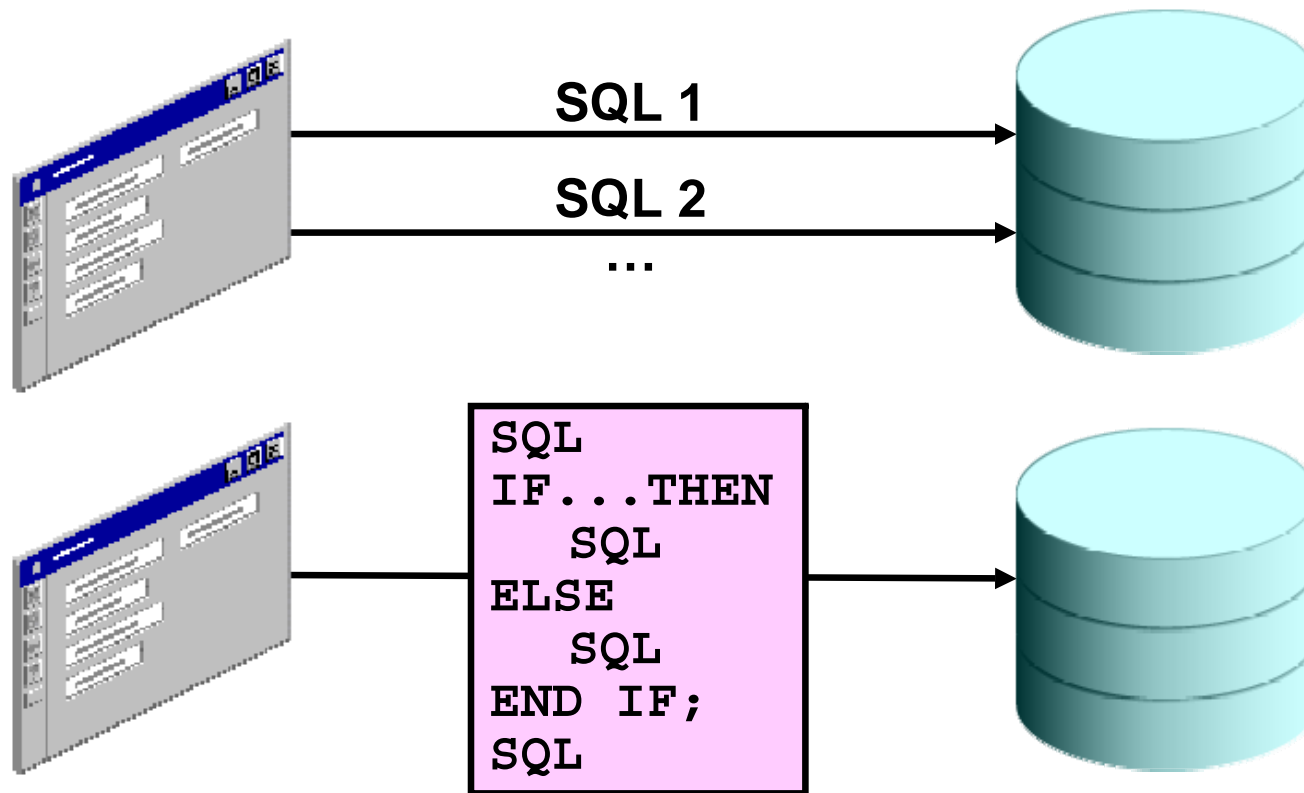
- Provides a block structure for executable units of code. Maintenance of code is made easier with such a well-defined structure.
- Provides procedural constructs such as:
  - Variables, constants, and types
  - Control structures such as conditional statements and loops
  - Reusable program units that are written once and executed many times

# PL/SQL Environment



# Benefits of PL/SQL

- Integration of procedural constructs with SQL
- Improved performance

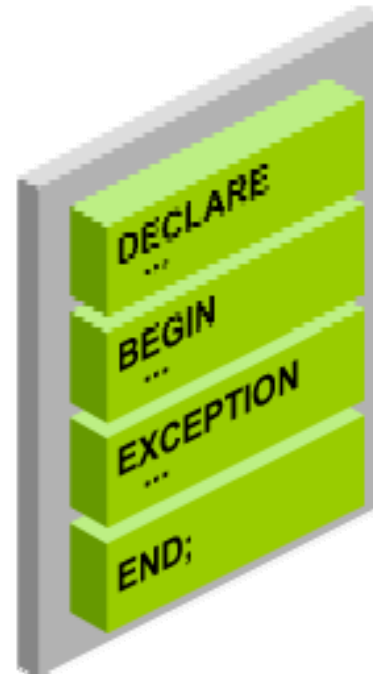


# Benefits of PL/SQL

- Modularized program development
- Integration with Oracle tools
- Portability
- Exception handling

# PL/SQL Block Structure

- DECLARE (optional)
  - Variables, cursors, user-defined exceptions
- BEGIN (mandatory)
  - SQL statements
  - PL/SQL statements
- EXCEPTION (optional)
  - Actions to perform when errors occur
- END; (mandatory)



# Block Types

## Anonymous

```
[DECLARE]

BEGIN
    --statements

[EXCEPTION]

END;
```

## Procedure

```
PROCEDURE name
IS

BEGIN
    --statements

[EXCEPTION]

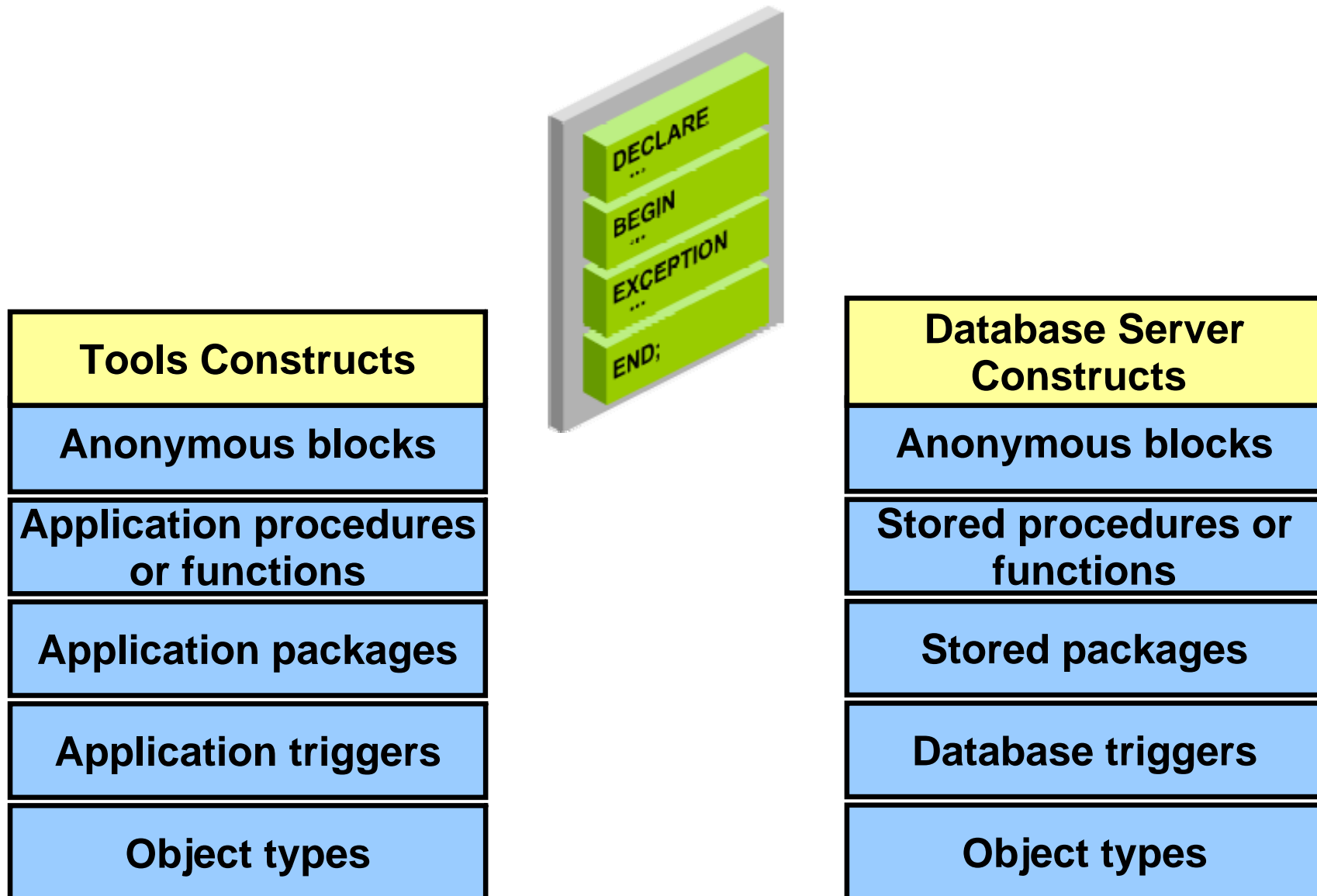
END;
```

## Function

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

END;
```

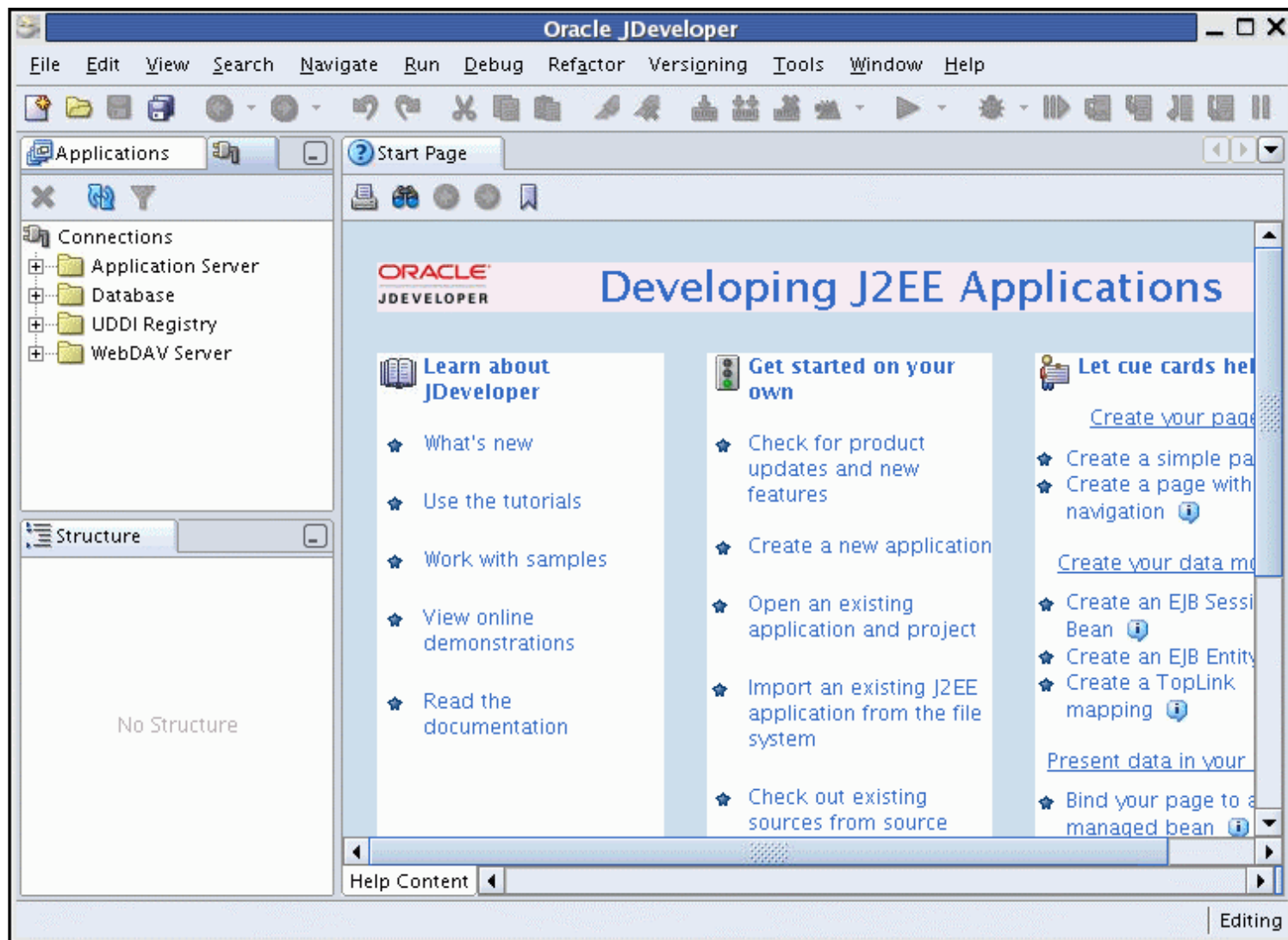
# Program Constructs





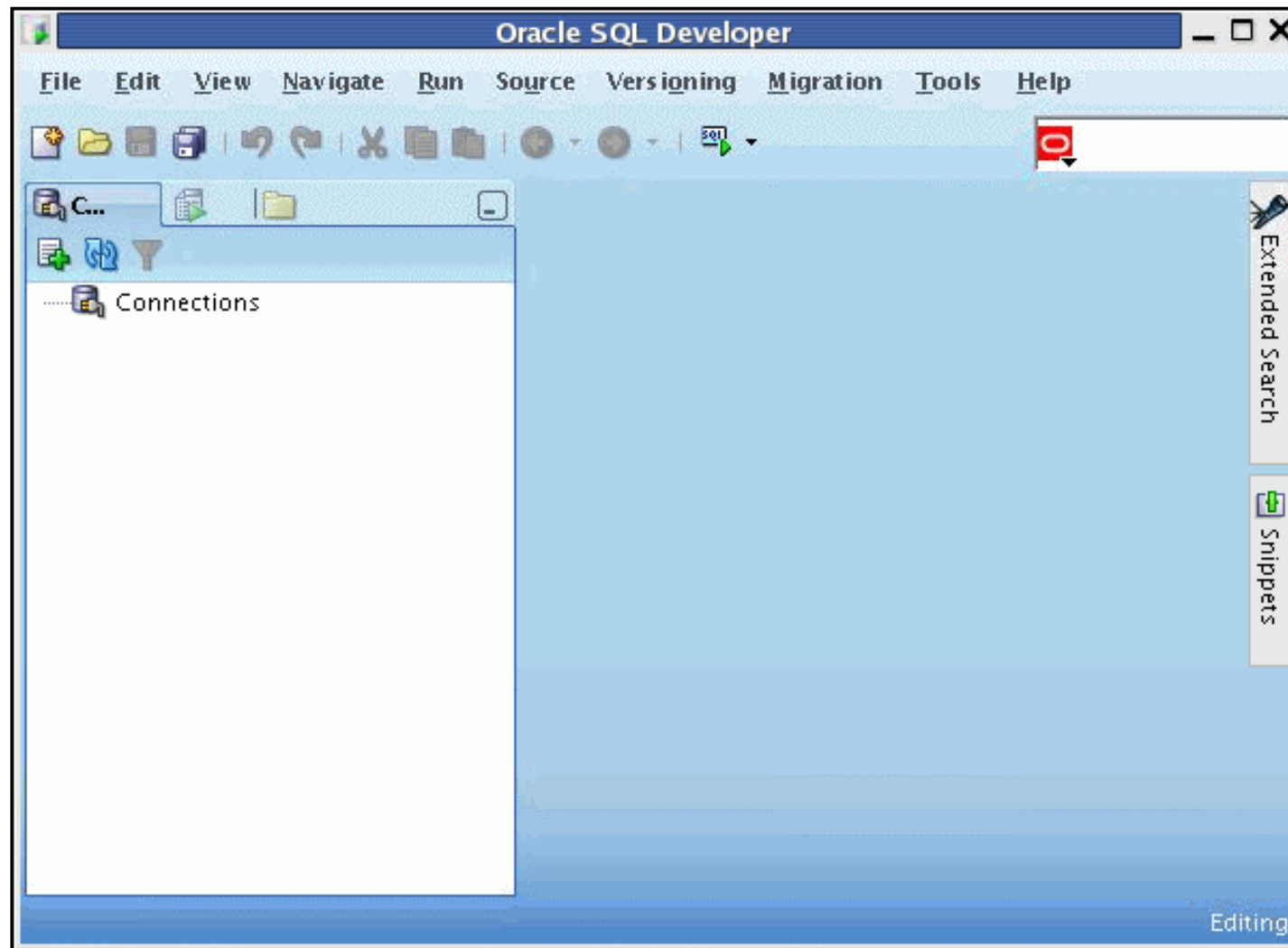
# PL/SQL Programming Environments

## Oracle JDeveloper

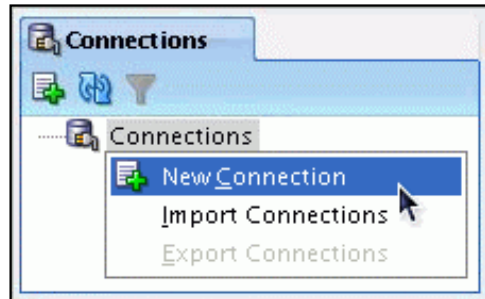


# PL/SQL Programming Environments

## Oracle SQL Developer



# Creating a Database Connection



The 'New / Select Database Connection' dialog box is shown. It contains the following fields and options:

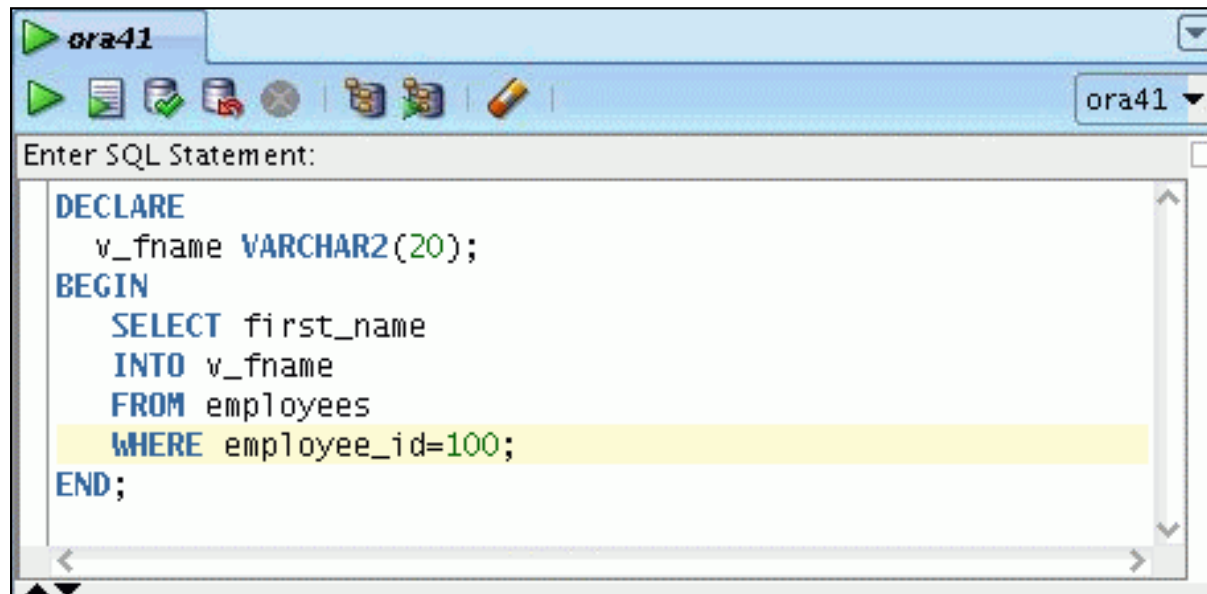
- Connection Name: ora41
- Username: ora41
- Password: \*\*\*\*\*
- ☒ Save Password
- Oracle
- Role: default
- Connection Type: Basic
- ☐ OS Authentication
- ☐ Proxy Connection
- Hostname: localhost
- Port: 1521
- ☒ SID
- ☐ Service name
- orcl

Status: Success

Buttons: Help, Save, Clear, Test, Connect, Cancel

# Creating an Anonymous Block

Enter the anonymous block in the SQL Developer workspace:

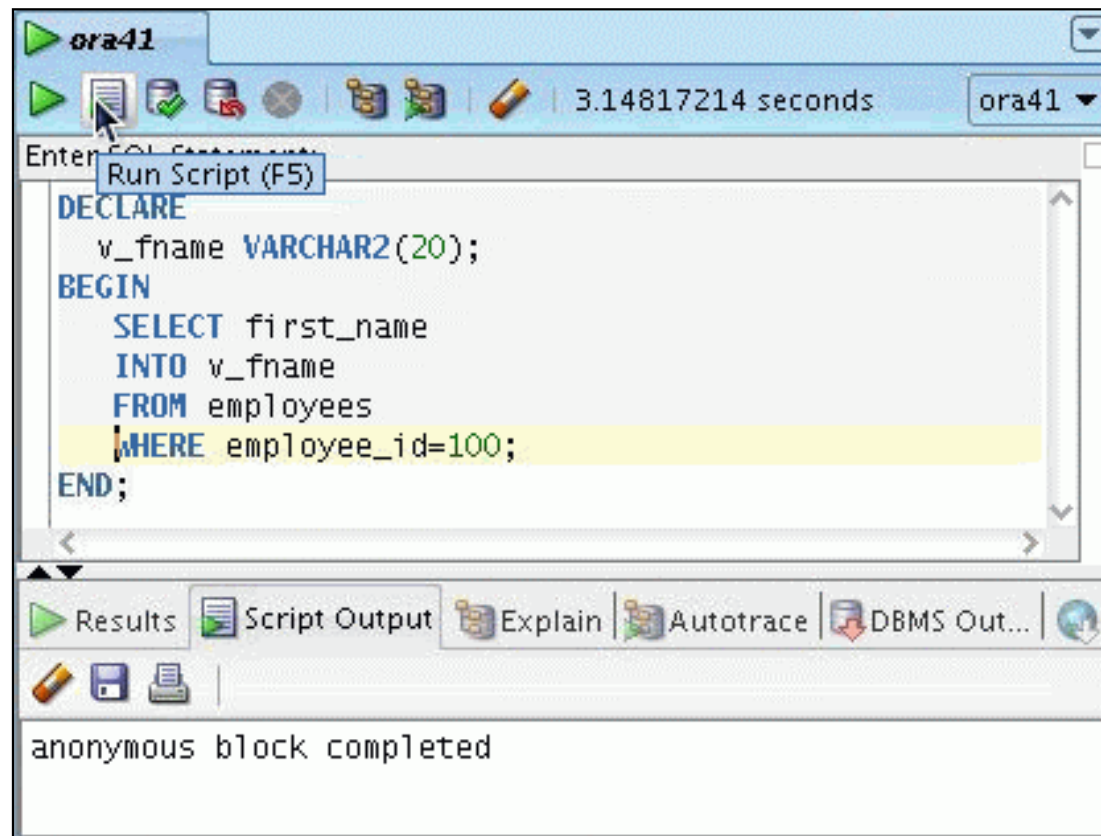
A screenshot of the SQL Developer workspace. The window title is 'ora41'. The toolbar shows various icons for execution, saving, and editing. The main text area is titled 'Enter SQL Statement:'. It contains the following SQL code:

```
DECLARE
  v_fname VARCHAR2(20);
BEGIN
  SELECT first_name
  INTO v_fname
  FROM employees
  WHERE employee_id=100;
END;
```

The line 'WHERE employee\_id=100;' is highlighted in yellow. The window has a scrollbar on the right side.

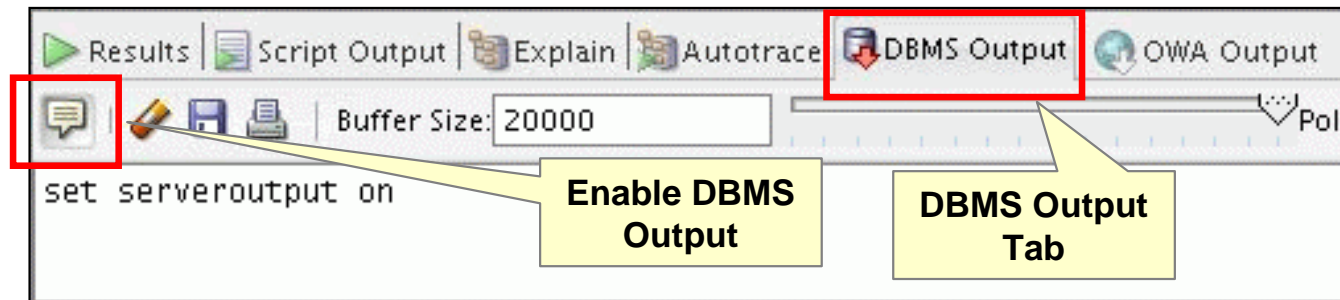
# Executing an Anonymous Block

Click the Run Script button to execute the anonymous block:



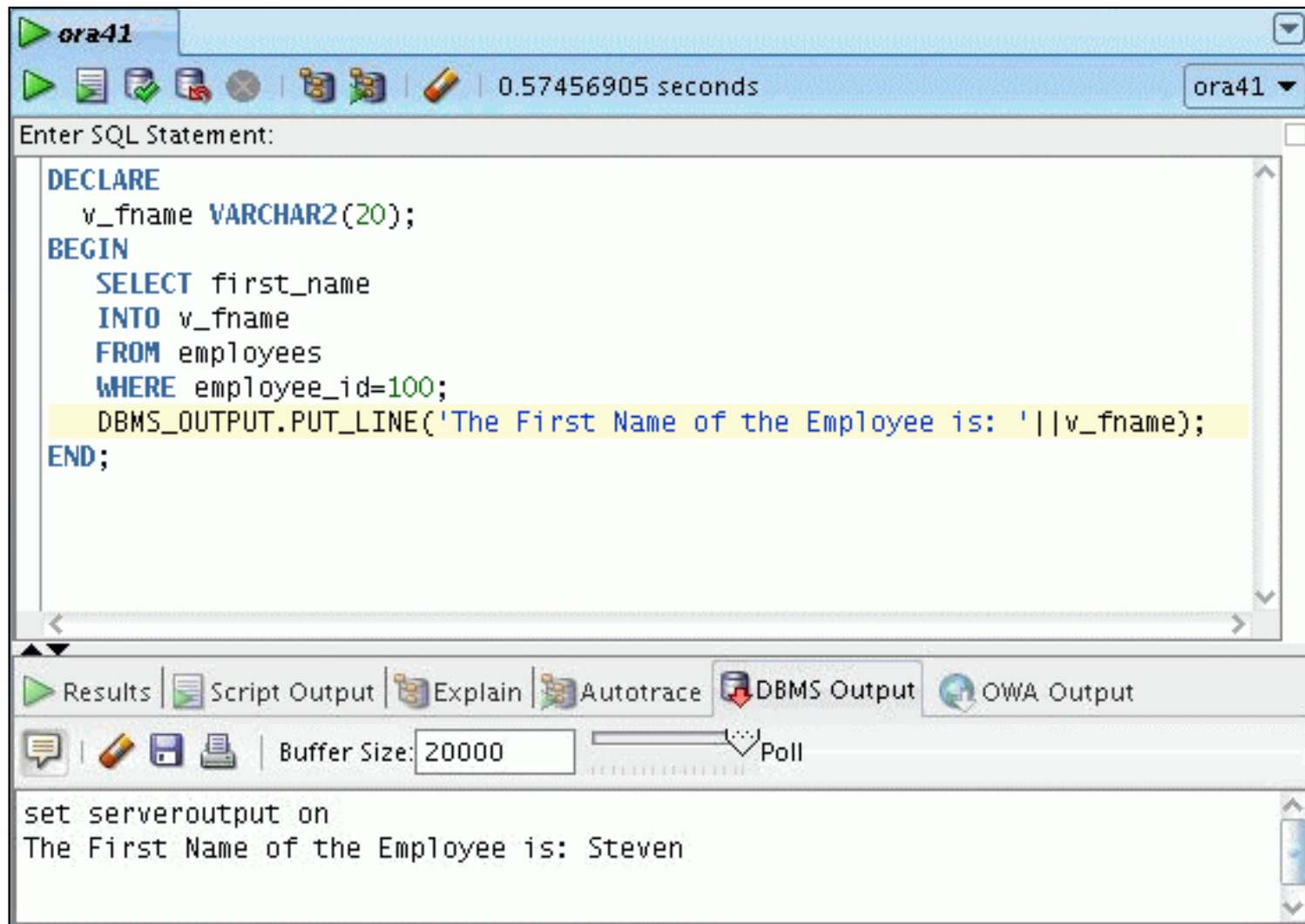
# Testing the Output of a PL/SQL Block

- Enable output in SQL Developer by clicking the Enable DBMS Output button on the DBMS Output tab:



- Use a predefined Oracle package and its procedure:
  - `DBMS_OUTPUT.PUT_LINE`

# Testing the Output of a PL/SQL Block



# Summary

In this lesson, you should have learned how to:

- Integrate SQL statements with PL/SQL program constructs
- Identify the benefits of PL/SQL
- Differentiate different PL/SQL block types
- Use Oracle SQL Developer as the programming environment for PL/SQL
- Output messages in PL/SQL



# Practice 1: Overview

This practice covers the following topics:

- Identifying which PL/SQL blocks execute successfully
- Creating and executing a simple PL/SQL block



# **Declaring PL/SQL Variables**

# Objectives

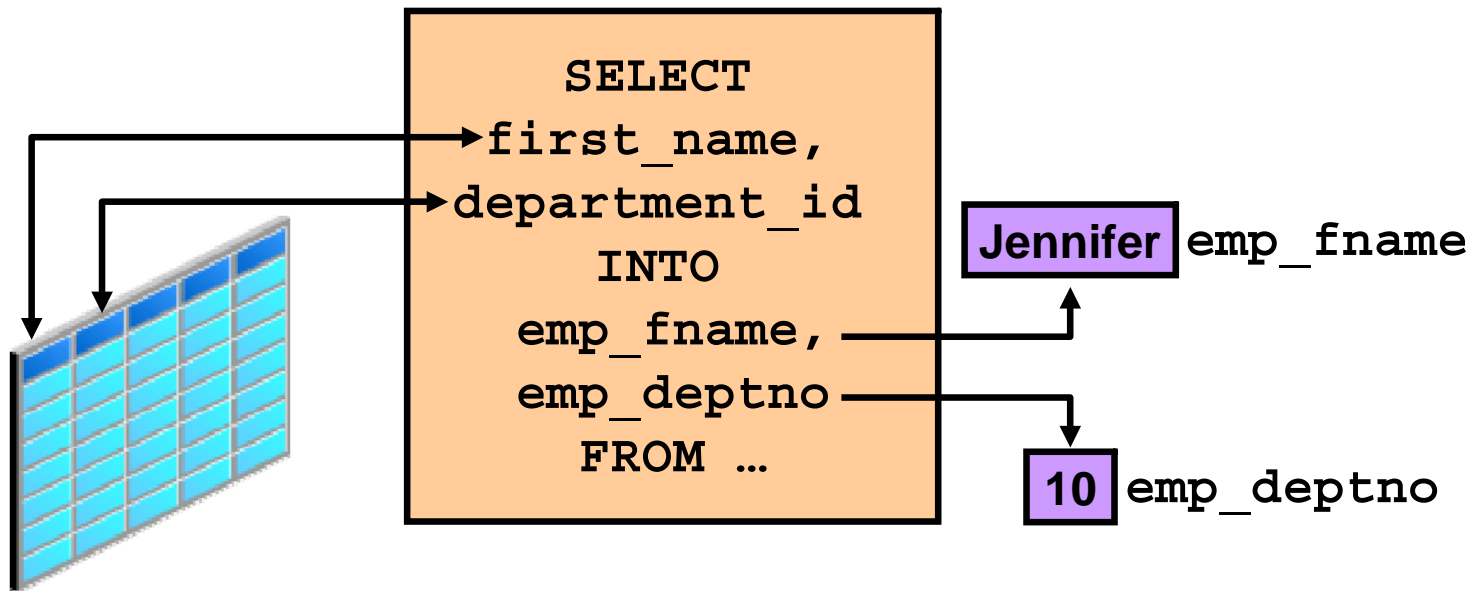
After completing this lesson, you should be able to do the following:

- Identify valid and invalid identifiers
- List the uses of variables
- Declare and initialize variables
- List and describe various data types
- Identify the benefits of using the `%TYPE` attribute
- Declare, use, and print bind variables

# Use of Variables

Variables can be used for:

- Temporary storage of data
- Manipulation of stored values
- Reusability



# Identifiers

Identifiers are used for:

- Naming a variable
- Providing conventions for variable names
  - Must start with a letter
  - Can include letters or numbers
  - Can include special characters (such as dollar sign, underscore, and pound sign)
  - Must limit the length to 30 characters
  - Must not be reserved words



# Handling Variables in PL/SQL

Variables are:

- Declared and initialized in the declarative section
- Used and assigned new values in the executable section
- Passed as parameters to PL/SQL subprograms
- Used to hold the output of a PL/SQL subprogram

# Declaring and Initializing PL/SQL Variables

## Syntax

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

## Examples

```
DECLARE  
    emp_hiredate    DATE;  
    emp_deptno      NUMBER(2) NOT NULL := 10;  
    location        VARCHAR2(13) := 'Atlanta';  
    c_comm          CONSTANT NUMBER := 1400;
```

# Declaring and Initializing PL/SQL Variables

1

```
SET SERVEROUTPUT ON
DECLARE
    Myname VARCHAR2(20);
BEGIN
    DBMS_OUTPUT.PUT_LINE('My name is: ' || Myname);
    Myname := 'John';
    DBMS_OUTPUT.PUT_LINE('My name is: ' || Myname);
END;
/
```

2

```
SET SERVEROUTPUT ON
DECLARE
    Myname VARCHAR2(20) := 'John';
BEGIN
    Myname := 'Steven';
    DBMS_OUTPUT.PUT_LINE('My name is: ' || Myname);
END;
/
```



# Delimiters in String Literals

```
SET SERVEROUTPUT ON
DECLARE
    event VARCHAR2(15);
BEGIN
    event := q'!Father's day!';
    DBMS_OUTPUT.PUT_LINE('3rd Sunday in June is :
    ' || event);
    event := q'[Mother's day]';
    DBMS_OUTPUT.PUT_LINE('2nd Sunday in May is :
    ' || event);
END;
/
```

```
anonymous block completed
3rd Sunday in June is : Father's day
2nd Sunday in May is : Mother's day
```

# Types of Variables

- PL/SQL variables:
  - Scalar
  - Composite
  - Reference
  - Large object (LOB)
- Non-PL/SQL variables: Bind variables

# Types of Variables

TRUE



25-JAN-01

**The soul of the lazy man  
desires, and he has nothing;  
but the soul of the diligent  
shall be made rich.**

256120.08



Atlanta

# Guidelines for Declaring and Initializing PL/SQL Variables

- Follow naming conventions.
- Use meaningful names for variables.
- Initialize variables designated as NOT NULL and CONSTANT.
- Initialize variables with the assignment operator (:=) or the DEFAULT keyword:

```
Myname VARCHAR2 (20) := 'John' ;
```

```
Myname VARCHAR2 (20) DEFAULT 'John' ;
```

- Declare one identifier per line for better readability and code maintenance.

# Guidelines for Declaring and Initializing PL/SQL Variables

- Avoid using column names as identifiers.

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

- Use the NOT NULL constraint when the variable must hold a value.

# Scalar Data Types

- Hold a single value
- Have no internal components

TRUE

25-JAN-01

**The soul of the lazy man  
desires, and he has nothing;  
but the soul of the diligent  
shall be made rich.**

256120.08

Atlanta

# Base Scalar Data Types

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

# Base Scalar Data Types

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



## **BINARY\_FLOAT and BINARY\_DOUBLE**

- Represent floating point numbers in IEEE 754 format
- Offer better interoperability and operational speed
- Store values beyond the values that the data type NUMBER can store
- Provide the benefits of closed arithmetic operations and transparent rounding

# Declaring Scalar Variables

## Examples

```
DECLARE
  emp_job          VARCHAR2 (9) ;
  count_loop       BINARY_INTEGER := 0;
  dept_total_sal   NUMBER (9,2)  := 0;
  orderdate        DATE := SYSDATE + 7;
  c_tax_rate       CONSTANT NUMBER (3,2) := 8.25;
  valid            BOOLEAN NOT NULL := TRUE;
  ...
```

# **%TYPE Attribute**

## The %TYPE attribute

- Is used to declare a variable according to:
  - A database column definition
  - Another declared variable
- Is prefixed with:
  - The database table and column
  - The name of the declared variable

# Declaring Variables with the %TYPE Attribute

## Syntax

```
identifier      table.column_name%TYPE;
```

## Examples

```
...  
  emp_lname      employees.last_name%TYPE;  
  balance        NUMBER(7,2);  
  min_balance    balance%TYPE := 1000;  
...
```

# Declaring Boolean Variables

- Only the values `TRUE`, `FALSE`, and `NULL` can be assigned to a Boolean variable.
- Conditional expressions use the logical operators `AND` and `OR` and the unary operator `NOT` to check the variable values.
- The variables always yield `TRUE`, `FALSE`, or `NULL`.
- Arithmetic, character, and date expressions can be used to return a Boolean value.

# Bind Variables

Bind variables are:

- Created in the environment
- Also called *host* variables
- Created with the `VARIABLE` keyword
- Used in SQL statements and PL/SQL blocks
- Accessed even after the PL/SQL block is executed
- Referenced with a preceding colon

# Printing Bind Variables

## Example

```
VARIABLE emp_salary NUMBER
BEGIN
    SELECT salary INTO :emp_salary
    FROM employees WHERE employee_id = 178;
END;
/
PRINT emp_salary
SELECT first_name, last_name FROM employees
WHERE salary=:emp_salary;
```

# Printing Bind Variables

## Example

```
VARIABLE emp_salary NUMBER
SET AUTOPRINT ON
BEGIN
    SELECT salary INTO :emp_salary
    FROM employees WHERE employee_id = 178;
END;
/
```

```
anonymous block completed
emp_salary
-----
7000
```

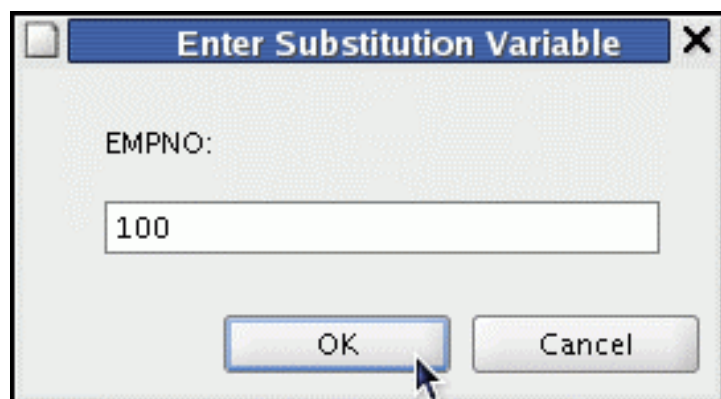


# Substitution Variables

- Are used to get user input at run time
- Are referenced within a PL/SQL block with a preceding ampersand
- Are used to avoid hard-coding values that can be obtained at run time

```
VARIABLE emp_salary NUMBER
SET AUTOPRINT ON
DECLARE
    empno NUMBER(6) := &empno;
BEGIN
    SELECT salary INTO :emp_salary
    FROM employees WHERE employee_id = empno;
END;
/
```

# Substitution Variables



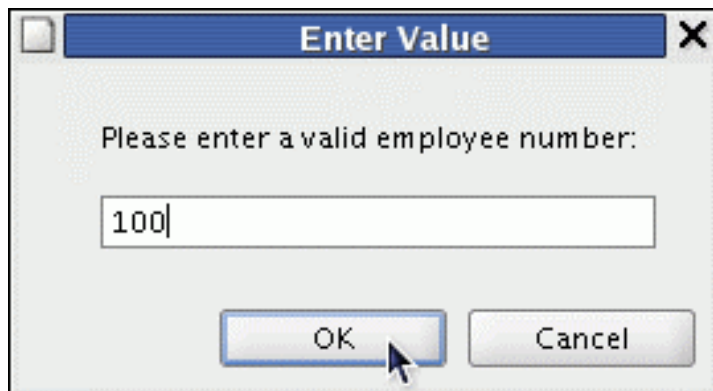
1

```
VARIABLE emp_salary NUMBER
SET AUTOPRINT ON
DECLARE
empno NUMBER(6):=100;
BEGIN
SELECT salary INTO :emp_salary
FROM employees WHERE employee_id = empno;
END;
anonymous block completed
emp_salary
-----
24000
```

2

# Prompt for Substitution Variables

```
SET VERIFY OFF
VARIABLE emp_salary NUMBER
ACCEPT empno PROMPT 'Please enter a valid employee
number: '
SET AUTOPRINT ON
DECLARE
    empno NUMBER(6) := &empno;
BEGIN
    SELECT salary INTO :emp_salary FROM employees
    WHERE employee_id = empno;
END;
/
```




# Using DEFINE for a User Variable

## Example

```
SET VERIFY OFF
DEFINE lname= Urman
DECLARE
    fname VARCHAR2(25);
BEGIN
    SELECT first_name INTO fname FROM employees
    WHERE last_name='&lname';
END;
/
```

# Composite Data Types

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

PL/SQL table structure

1	SMITH
2	JONES
3	NANCY
4	TIM

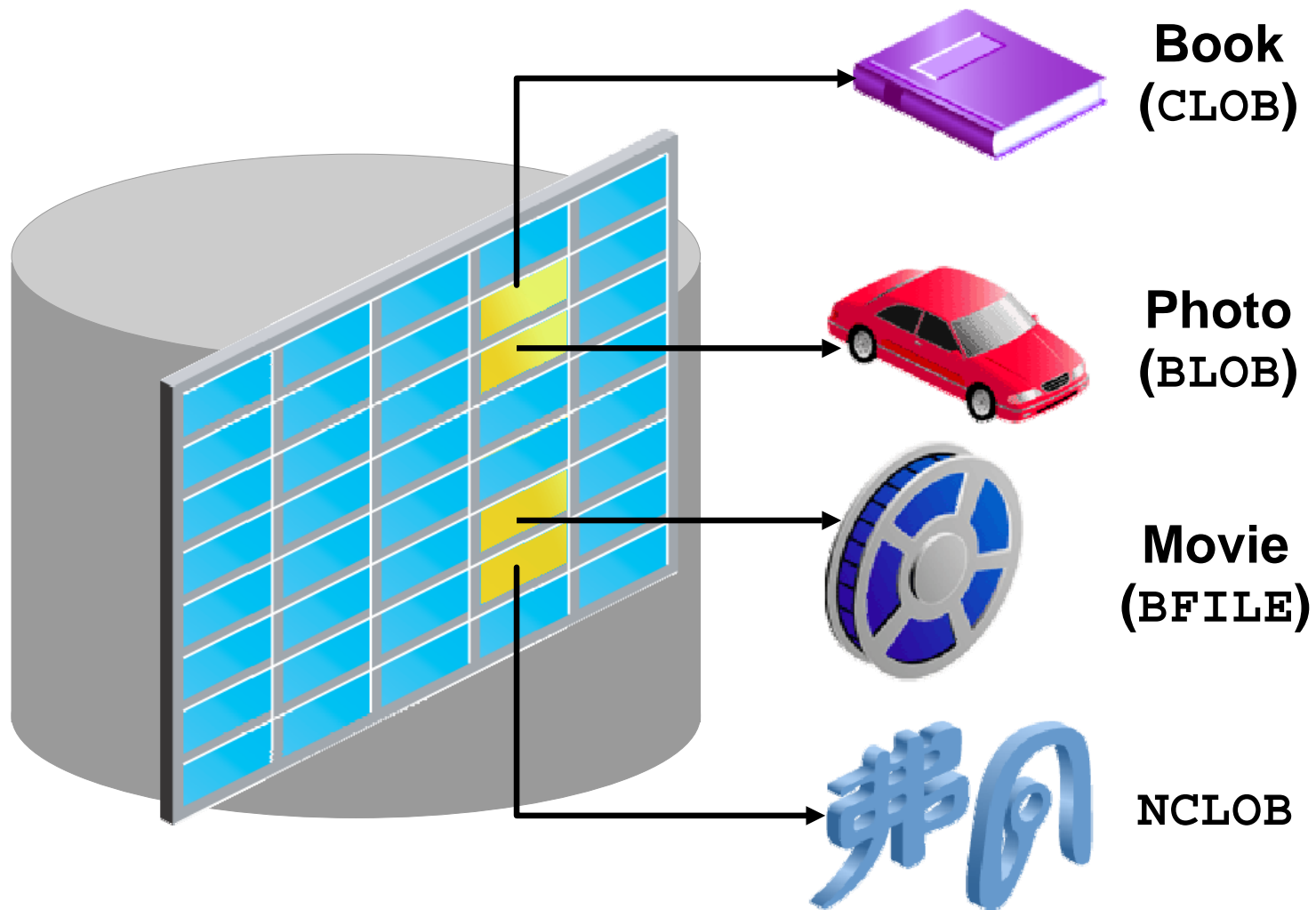
PLS\_INTEGER  
VARCHAR2

PL/SQL table structure

1	5000
2	2345
3	12
4	3456

PLS\_INTEGER  
NUMBER

# LOB Data Type Variables



# Summary

In this lesson, you should have learned how to:

- Recognize valid and invalid identifiers
- Declare variables in the declarative section of a PL/SQL block
- Initialize variables and use them in the executable section
- Differentiate between scalar and composite data types
- Use the %TYPE attribute
- Use bind variables

## Practice 2: Overview

This practice covers the following topics:

- Determining valid identifiers
- Determining valid variable declarations
- Declaring variables within an anonymous block
- Using the %TYPE attribute to declare variables
- Declaring and printing a bind variable
- Executing a PL/SQL block





# Writing Executable Statements

# Objectives

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

- Identify lexical units in a PL/SQL block
- Use built-in SQL functions in PL/SQL
- Describe when implicit conversions take place and when explicit conversions have to be dealt with
- Write nested blocks and qualify variables with labels
- Write readable code with appropriate indentations

# Lexical Units in a PL/SQL Block

Lexical units:

- Are building blocks of any PL/SQL block
- Are sequences of characters including letters, numerals, tabs, spaces, returns, and symbols
- Can be classified as:
  - Identifiers
  - Delimiters
  - Literals
  - Comments

# PL/SQL Block Syntax and Guidelines

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

```
name := 'Henderson';
```

- Numbers can be simple values or scientific notation.
- Statements can continue over several lines.

# Commenting Code

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

## Example

```
DECLARE
...
annual_sal NUMBER (9,2);
BEGIN    -- Begin the executable section

/* Compute the annual salary based on the
   monthly salary input from the user */
annual_sal := monthly_sal * 12;
END;    -- This is the end of the block
/
```

# SQL Functions in PL/SQL

- Available in procedural statements:
  - Single-row number
  - Single-row character
  - Data type conversion
  - Date
  - Timestamp
  - GREATEST and LEAST
  - Miscellaneous functions
- Not available in procedural statements:
  - DECODE
  - Group functions

# SQL Functions in PL/SQL: Examples

- Get the length of a string:

```
desc_size INTEGER(5);  
prod_description VARCHAR2(70):='You can use this  
product with your radios for higher frequency';  
  
-- get the length of the string in prod_description  
desc_size:= LENGTH(prod_description);
```

- Convert the employee name to lowercase:

```
emp_name:= LOWER(emp_name);
```

# Data Type Conversions

- Convert data to comparable data types
- Are of two types:
  - Implicit conversions
  - Explicit conversions
- Some conversion functions:
  - TO\_CHAR
  - TO\_DATE
  - TO\_NUMBER
  - TO\_TIMESTAMP



# Data Type Conversion

1

```
date_of_joining DATE:= '02-Feb-2000';
```

2

```
date_of_joining DATE:= 'February 02,2000';
```

3

```
date_of_joining DATE:= TO_DATE('February  
02,2000','Month DD, YYYY');
```

# Nested Blocks

PL/SQL blocks can be nested.

- An executable section (BEGIN ... END) can contain nested blocks.
- An exception section can contain nested blocks.



# Nested Blocks

Example:

```
DECLARE
  outer_variable VARCHAR2(20) := 'GLOBAL VARIABLE';
BEGIN
  DECLARE
    inner_variable VARCHAR2(20) := 'LOCAL VARIABLE';
  BEGIN
    DBMS_OUTPUT.PUT_LINE(inner_variable);
    DBMS_OUTPUT.PUT_LINE(outer_variable);
  END;
  DBMS_OUTPUT.PUT_LINE(outer_variable);
END;
/
```

# Variable Scope and Visibility

```
DECLARE
  father_name VARCHAR2(20):='Patrick';
  date_of_birth DATE:='20-Apr-1972';
BEGIN
  DECLARE
    child_name VARCHAR2(20):='Mike';
    date_of_birth DATE:='12-Dec-2002';
  BEGIN
    DBMS_OUTPUT.PUT_LINE('Father''s Name: ' || father_name);
    DBMS_OUTPUT.PUT_LINE('Date of Birth: ' || date_of_birth);
    DBMS_OUTPUT.PUT_LINE('Child''s Name: ' || child_name);
  END;
  DBMS_OUTPUT.PUT_LINE('Date of Birth: ' || date_of_birth);
END;
/
```

# Qualify an Identifier

```
BEGIN <<outer>>
DECLARE
  father_name VARCHAR2(20):='Patrick';
  date_of_birth DATE:='20-Apr-1972';
BEGIN
  DECLARE
    child_name VARCHAR2(20):='Mike';
    date_of_birth DATE:='12-Dec-2002';
  BEGIN
    DBMS_OUTPUT.PUT_LINE('Father's Name: ' || father_name);
    DBMS_OUTPUT.PUT_LINE('Date of Birth: '
                        || outer.date_of_birth);
    DBMS_OUTPUT.PUT_LINE('Child's Name: ' || child_name);
  END;
  DBMS_OUTPUT.PUT_LINE('Date of Birth: ' || date_of_birth);
END;
END outer;
```


# Determining Variable Scope

```
BEGIN <<outer>>
DECLARE
    sal          NUMBER(7,2) := 60000;
    comm         NUMBER(7,2) := sal * 0.20;
    message      VARCHAR2(255) := ' eligible for commission';
BEGIN
    DECLARE
        sal          NUMBER(7,2) := 50000;
        comm         NUMBER(7,2) := 0;
        total_comp   NUMBER(7,2) := sal + comm;
    BEGIN
        message := 'CLERK not' || message;
        outer.comm := sal * 0.30;
    END;
    message := 'SALESMAN' || message;
END;
END outer;
/
```

1

2

# Operators in PL/SQL

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

# Operators in PL/SQL

## Examples:

- Increment the counter for a loop.

```
loop_count := loop_count + 1;
```

- Set the value of a Boolean flag.

```
good_sal := sal BETWEEN 50000 AND 150000;
```

- Validate whether an employee number contains a value.

```
valid := (empno IS NOT NULL);
```



# Programming Guidelines

Make code maintenance easier by:

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

# Indenting Code

For clarity, indent each level of code.

Example:

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

```
DECLARE
  deptno          NUMBER(4);
  location_id     NUMBER(4);
BEGIN
  SELECT  department_id,
          location_id
  INTO    deptno,
          location_id
  FROM    departments
  WHERE   department_name
          = 'Sales';

  ...
END;
/
```

# Summary

In this lesson, you should have learned how to:

- Use built-in SQL functions in PL/SQL
- Write nested blocks to break logically related functionalities
- Decide when to perform explicit conversions
- Qualify variables in nested blocks

# Practice 3: Overview

This practice covers the following topics:

- Reviewing scoping and nesting rules
- Writing and testing PL/SQL blocks

# 4

## Interacting with the Oracle Server

# Objectives

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

- Determine which SQL statements can be directly included in a PL/SQL executable block
- Manipulate data with DML statements in PL/SQL
- Use transaction control statements in PL/SQL
- Use the `INTO` clause to hold the values returned by a SQL statement
- Differentiate between implicit cursors and explicit cursors
- Use SQL cursor attributes

# SQL Statements in PL/SQL

- Retrieve a row from the database by using the `SELECT` command.
- Make changes to rows in the database by using DML commands.
- Control a transaction with the `COMMIT`, `ROLLBACK`, or `SAVEPOINT` command.

# SELECT Statements in PL/SQL

Retrieve data from the database with a `SELECT` statement.

Syntax:

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



# SELECT Statements in PL/SQL

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

## Example

```
SET SERVEROUTPUT ON
DECLARE
  fname VARCHAR2(25);
BEGIN
  SELECT first_name INTO fname
  FROM employees WHERE employee_id=200;
  DBMS_OUTPUT.PUT_LINE(' First Name is : ' || fname);
END;
/
```

# Retrieving Data in PL/SQL

Retrieve `hire_date` and `salary` for the specified employee.

Example:

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

# Retrieving Data in PL/SQL

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

Example:

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

# Naming Conventions

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

Error report:

ORA-01422: exact fetch returns more than requested number of rows

ORA-06512: at line 6

01422. 00000 - "exact fetch returns more than requested number of rows"

\*Cause: The number specified in exact fetch is less than the rows returned.

\*Action: Rewrite the query or change number of rows requested

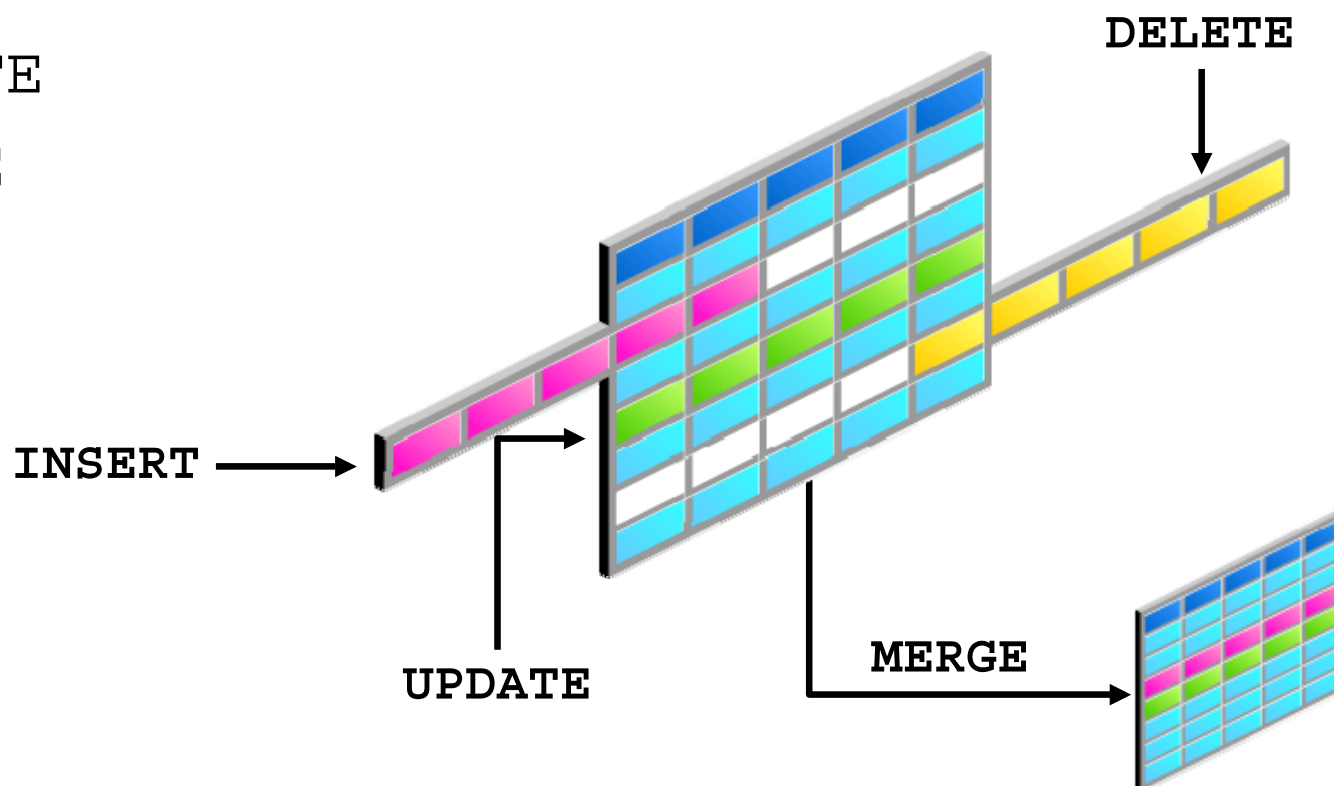
# Naming Conventions

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

# Manipulating Data Using PL/SQL

Make changes to database tables by using DML commands:

- INSERT
- UPDATE
- DELETE
- MERGE



# Inserting Data

Add new employee information to the EMPLOYEES table.

Example:

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

# Updating Data

Increase the salary of all employees who are stock clerks.

Example:

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



# Deleting Data

Delete rows that belong to department 10 from the `employees` table.

Example:

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

# Merging Rows

Insert or update rows in the `copy_emp` table to match the `employees` table.

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

# SQL Cursor

- A cursor is a pointer to the private memory area allocated by the Oracle server.
- There are two types of cursors:
  - Implicit: Created and managed internally by the Oracle server to process SQL statements
  - Explicit: Explicitly declared by the programmer

# SQL Cursor Attributes for Implicit Cursors

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

<b>SQL%FOUND</b>	Boolean attribute that evaluates to <code>TRUE</code> if the most recent SQL statement returned at least one row
<b>SQL%NOTFOUND</b>	Boolean attribute that evaluates to <code>TRUE</code> if the most recent SQL statement did not return even one row
<b>SQL%ROWCOUNT</b>	An integer value that represents the number of rows affected by the most recent SQL statement

# SQL Cursor Attributes for Implicit Cursors

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

## Example

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

```
END;
/
PRINT rows_deleted
```

# Summary

In this lesson, you should have learned how to:

- Embed DML statements, transaction control statements, and DDL statements in PL/SQL
- Use the `INTO` clause, which is mandatory for all `SELECT` statements in PL/SQL
- Differentiate between implicit cursors and explicit cursors
- Use SQL cursor attributes to determine the outcome of SQL statements

# Practice 4: Overview

This practice covers the following topics:

- Selecting data from a table
- Inserting data into a table
- Updating data in a table
- Deleting a record from a table

# 5

## Writing Control Structures

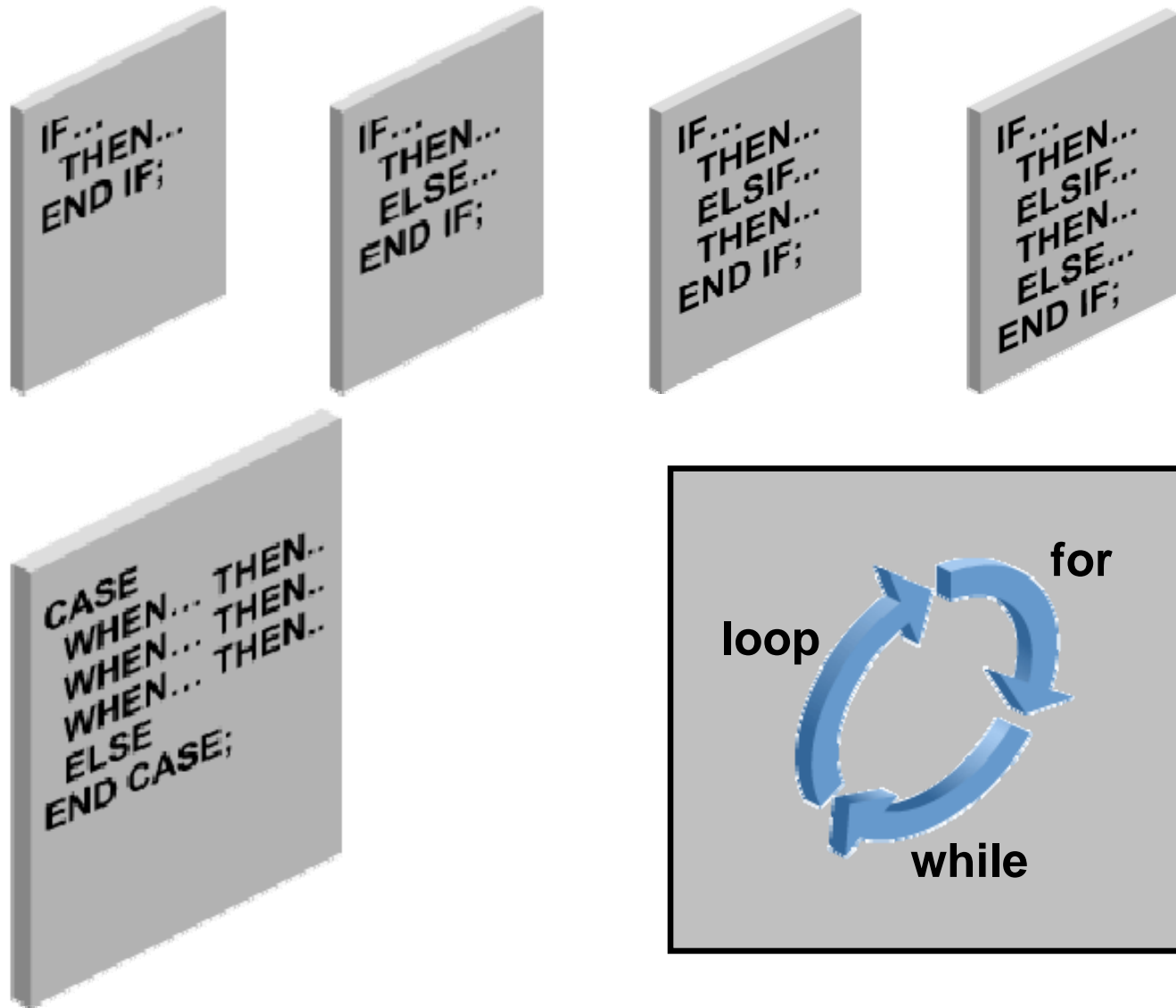


# Objectives

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

- Identify the uses and types of control structures
- Construct an `IF` statement
- Use `CASE` statements and `CASE` expressions
- Construct and identify different loop statements
- Use guidelines when using conditional control structures

# Controlling Flow of Execution



# IF Statements

Syntax:

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

# Simple IF Statement

```
DECLARE
  myage number:=31;
BEGIN
  IF myage < 11
  THEN
    DBMS_OUTPUT.PUT_LINE(' I am a child ');
  END IF;
END;
/
```

```
anonymous block completed
```

# IF THEN ELSE Statement

```
SET SERVEROUTPUT ON
DECLARE
myage number:=31;
BEGIN
IF myage < 11
  THEN
    DBMS_OUTPUT.PUT_LINE(' I am a child ');
  ELSE
    DBMS_OUTPUT.PUT_LINE(' I am not a child ');
END IF;
END;
/
```

```
anonymous block completed
I am not a child
```

# IF ELSIF ELSE Clause

```
DECLARE
myage number:=31;
BEGIN
IF myage < 11
THEN
    DBMS_OUTPUT.PUT_LINE(' I am a child ');
ELSIF myage < 20
THEN
    DBMS_OUTPUT.PUT_LINE(' I am young ');
ELSIF myage < 30
THEN
    DBMS_OUTPUT.PUT_LINE(' I am in my twenties');
ELSIF myage < 40
THEN
    DBMS_OUTPUT.PUT_LINE(' I am in my thirties');
ELSE
    DBMS_OUTPUT.PUT_LINE(' I am always young ');
END IF;
END;
/
```

```
anonymous block completed
I am in my thirties
```

# NULL Values in IF Statements

```
DECLARE
myage number;
BEGIN
IF myage < 11
  THEN
    DBMS_OUTPUT.PUT_LINE(' I am a child ');
  ELSE
    DBMS_OUTPUT.PUT_LINE(' I am not a child ');
END IF;
END;
/
```

```
anonymous block completed
I am not a child
```

# CASE Expressions

- A CASE expression selects a result and returns it.
- To select the result, the CASE expression uses expressions. The value returned by these expressions is used to select one of several alternatives.

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



# CASE Expressions: Example

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

# Searched CASE Expressions

```
DECLARE
    grade CHAR(1) := UPPER('&grade');
    appraisal VARCHAR2(20);
BEGIN
    appraisal :=
        CASE
            WHEN grade = 'A' THEN 'Excellent'
            WHEN grade IN ('B','C') THEN 'Good'
            ELSE 'No such grade'
        END;
    DBMS_OUTPUT.PUT_LINE ('Grade: ' || grade || '
                          Appraisal ' || appraisal);
END;
/
```

# CASE Statement

```
DECLARE
    deptid NUMBER;
    deptname VARCHAR2(20);
    emps NUMBER;
    mngid NUMBER:= 108;
BEGIN
    CASE mngid
        WHEN 108 THEN
            SELECT department_id, department_name
            INTO deptid, deptname FROM departments
            WHERE manager_id=108;
            SELECT count(*) INTO emps FROM employees
            WHERE department_id=deptid;
        WHEN 200 THEN
            ...
        END CASE;
    DBMS_OUTPUT.PUT_LINE ('You are working in the ' || deptname ||
    ' department. There are ' || emps || ' employees in this
    department');
END;
/
```

# Handling Nulls

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

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

# Logic Tables

Build a simple Boolean condition with a comparison operator.

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

# Boolean Conditions

What is the value of `flag` in each case?

```
flag := reorder_flag AND available_flag;
```

REORDER_FLAG	AVAILABLE_FLAG	FLAG
TRUE	TRUE	? (1)
TRUE	FALSE	? (2)
NULL	TRUE	? (3)
NULL	FALSE	? (4)

# Iterative Control: LOOP Statements

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



# Basic Loops

Syntax:

```
LOOP  
  statement1;  
  . . .  
  EXIT [WHEN condition];  
END LOOP;
```



# Basic Loops

## Example

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

# WHILE Loops

Syntax:

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

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

# WHILE Loops

## Example

```
DECLARE
    countryid    locations.country_id%TYPE := 'CA';
    loc_id       locations.location_id%TYPE;
    new_city     locations.city%TYPE := 'Montreal';
    counter      NUMBER := 1;
BEGIN
    SELECT MAX(location_id) INTO loc_id FROM locations
    WHERE country_id = countryid;
    WHILE counter <= 3 LOOP
        INSERT INTO locations(location_id, city, country_id)
        VALUES((loc_id + counter), new_city, countryid);
        counter := counter + 1;
    END LOOP;
END;
/
```

# FOR Loops

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

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

# FOR Loops

Example:

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

# FOR Loops

## Guidelines

- Reference the counter within the loop only; it is undefined outside the loop.
- Do not reference the counter as the target of an assignment.
- Neither loop bound should be NULL.

# Guidelines for Loops

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

# Nested Loops and Labels

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



# Nested Loops and Labels

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

# Summary

In this lesson, you should have learned how to change the logical flow of statements by using the following control structures:

- Conditional (`IF` statement)
- `CASE` expressions and `CASE` statements
- Loops:
  - Basic loop
  - `FOR` loop
  - `WHILE` loop
- `EXIT` statements

# Practice 5: Overview

This practice covers the following topics:

- Performing conditional actions by using the `IF` statement
- Performing iterative steps by using the loop structure



# **Working with Composite Data Types**

# Objectives

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

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

# Composite Data Types

- Can hold multiple values (unlike scalar types)
- Are of two types:
  - PL/SQL records
  - PL/SQL collections
    - INDEX BY tables or associative arrays
    - Nested table
    - VARRAY

# Composite Data Types

- Use PL/SQL records when you want to store values of different data types but only one occurrence at a time.
- Use PL/SQL collections when you want to store values of the same data type.

# PL/SQL Records

- Must contain one or more components (called *fields*) of any scalar, RECORD, or INDEX BY table data type
- Are similar to structures in most 3GL languages (including C and C++)
- Are user defined and can be a subset of a row in a table
- Treat a collection of fields as a logical unit
- Are convenient for fetching a row of data from a table for processing



# Creating a PL/SQL Record

Syntax:

1

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

2

```
identifier    type_name;
```

*field\_declaration*:

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

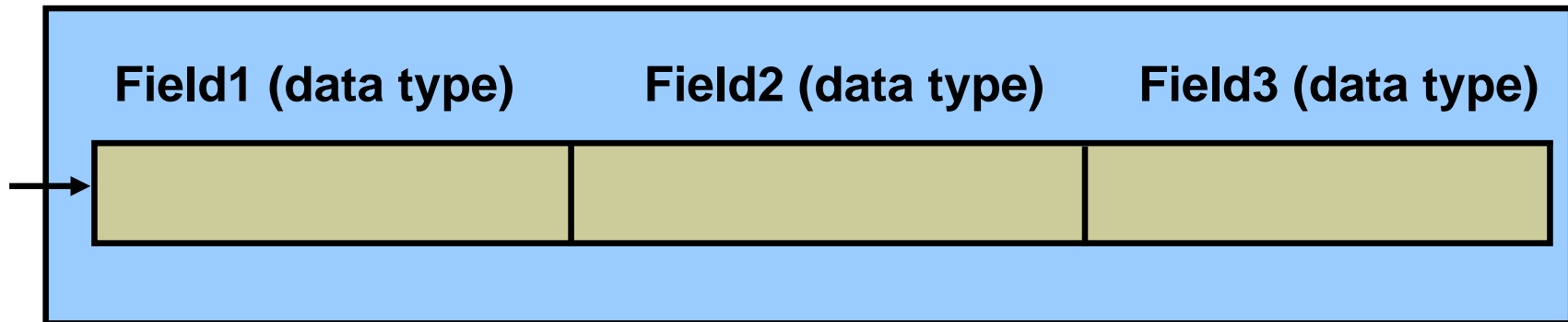
# Creating a PL/SQL Record

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

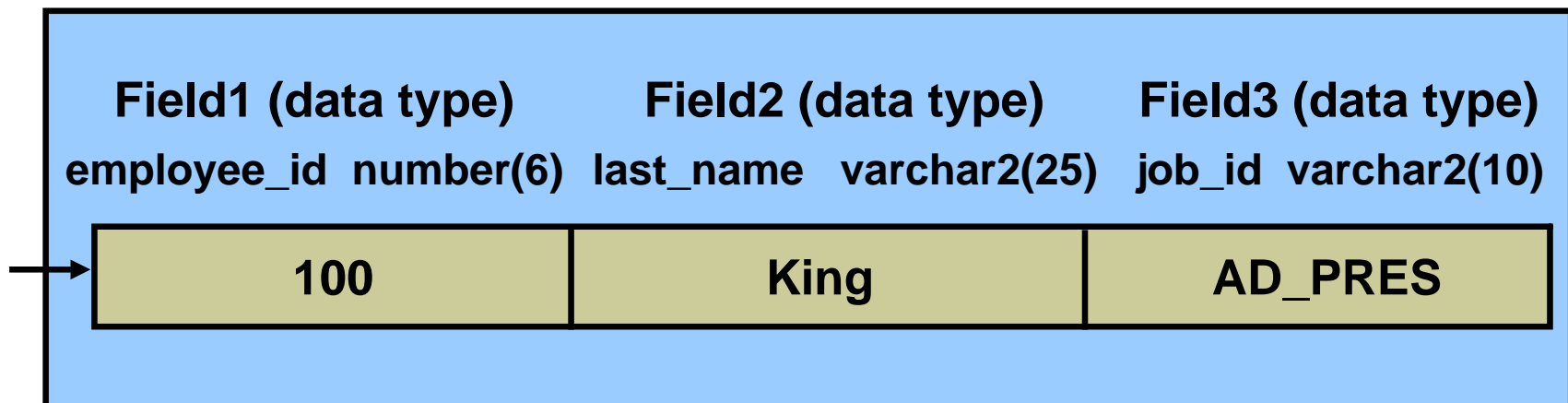
Example:

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

# PL/SQL Record Structure



Example:



## **%ROWTYPE Attribute**

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

Syntax:

```
DECLARE  
    identifier reference%ROWTYPE;
```

# Advantages of Using %ROWTYPE

- The number and data types of the underlying database columns need not be known—and in fact might change at run time.
- The %ROWTYPE attribute is useful when retrieving a row with the `SELECT *` statement.

# %ROWTYPE Attribute

```
...  
DEFINE employee_number = 124  
DECLARE  
    emp_rec    employees%ROWTYPE;  
BEGIN  
    SELECT * INTO emp_rec FROM employees  
    WHERE   employee_id = &employee_number;  
    INSERT INTO retired_ems(empno, ename, job, mgr,  
        hiredate, leavedate, sal, comm, deptno)  
    VALUES (emp_rec.employee_id, emp_rec.last_name,  
        emp_rec.job_id, emp_rec.manager_id,  
        emp_rec.hire_date, SYSDATE, emp_rec.salary,  
        emp_rec.commission_pct, emp_rec.department_id);  
END;  
/
```

# Inserting a Record by Using %ROWTYPE

```
...  
DEFINE employee_number = 124  
DECLARE  
    emp_rec  retired_emps%ROWTYPE;  
BEGIN  
    SELECT employee_id, last_name, job_id, manager_id,  
           hire_date, hire_date, salary, commission_pct,  
           department_id INTO emp_rec FROM employees  
    WHERE  employee_id = &employee_number;  
    INSERT INTO retired_emps VALUES emp_rec;  
END;  
/  
SELECT * FROM retired_emps;
```

# Updating a Row in a Table by Using a Record

```
SET SERVEROUTPUT ON
SET VERIFY OFF
DEFINE employee_number = 124
DECLARE
    emp_rec retired_emps%ROWTYPE;
BEGIN
    SELECT * INTO emp_rec FROM retired_emps;
    emp_rec.leavedate:=SYSDATE;
    UPDATE retired_emps SET ROW = emp_rec WHERE
        empno=&employee_number;
END;
/
SELECT * FROM retired_emps;
```



# INDEX BY Tables or Associative Arrays

- Are PL/SQL structures with two columns:
  - Primary key of integer or string data type
  - Column of scalar or record data type
- Are unconstrained in size. However, the size depends on the values that the key data type can hold.

# Creating an INDEX BY Table

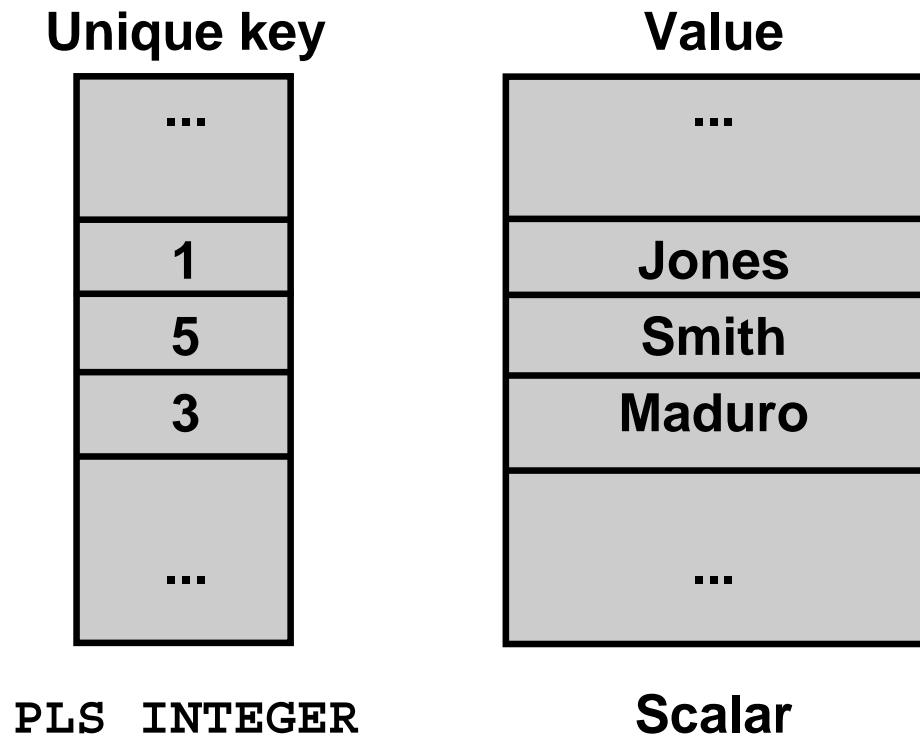
Syntax:

```
TYPE type_name IS TABLE OF
    {column_type | variable%TYPE
    | table.column%TYPE} [NOT NULL]
    | table%ROWTYPE
    [INDEX BY PLS_INTEGER | BINARY_INTEGER
    | VARCHAR2(<size>)];
identifier    type_name;
```

Declare an INDEX BY table to store the last names of employees:

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

# INDEX BY Table Structure



# Creating an INDEX BY Table

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

# Using INDEX BY Table Methods

The following methods make INDEX BY tables easier to use:

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

# INDEX BY Table of Records

Define an INDEX BY table variable to hold an entire row from a table.

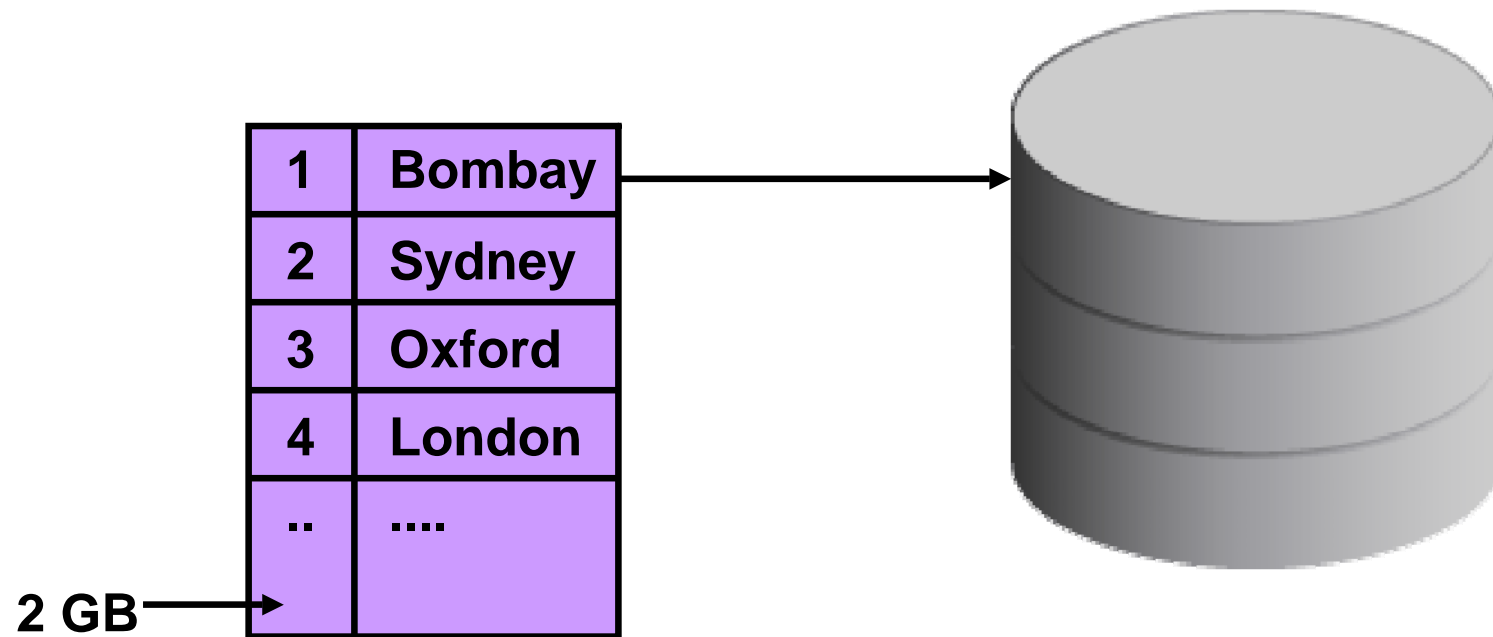
Example:

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

# INDEX BY Table of Records: Example

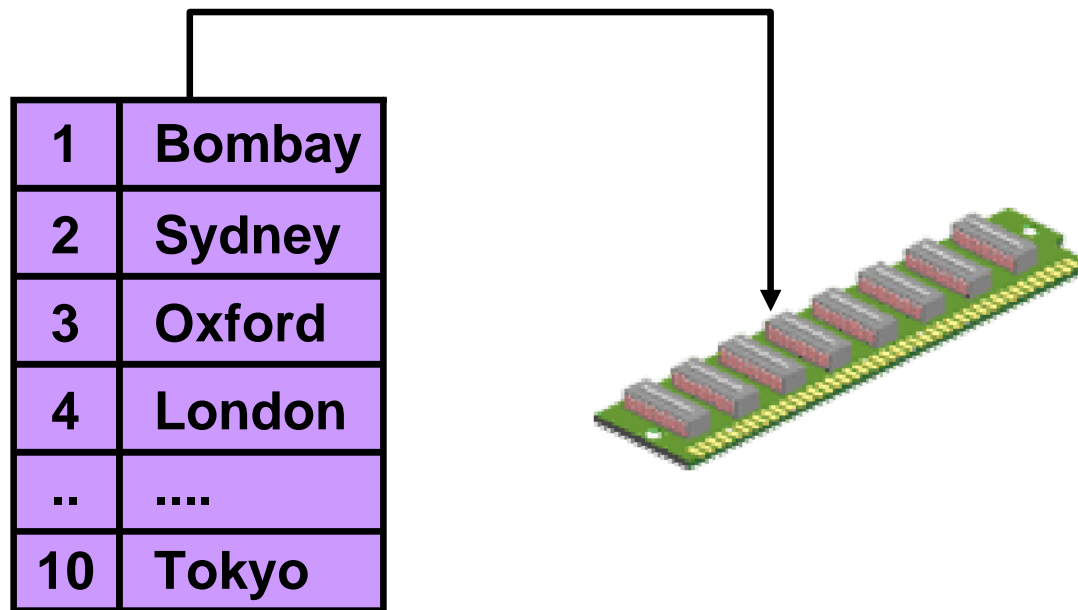
```
SET SERVEROUTPUT ON
DECLARE
    TYPE emp_table_type IS TABLE OF
        employees%ROWTYPE INDEX BY PLS_INTEGER;
    my_emp_table    emp_table_type;
    max_count       NUMBER(3) := 104;
BEGIN
    FOR i IN 100..max_count
    LOOP
        SELECT * INTO my_emp_table(i) FROM employees
        WHERE employee_id = i;
    END LOOP;
    FOR i IN my_emp_table.FIRST..my_emp_table.LAST
    LOOP
        DBMS_OUTPUT.PUT_LINE(my_emp_table(i).last_name);
    END LOOP;
END;
/
```

# Nested Tables





# VARRAY



# Summary

In this lesson, you should have learned how to:

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

# Practice 6: Overview

This practice covers the following topics:

- Declaring `INDEX BY` tables
- Processing data by using `INDEX BY` tables
- Declaring a PL/SQL record
- Processing data by using a PL/SQL record



# Using Explicit Cursors

# Objectives

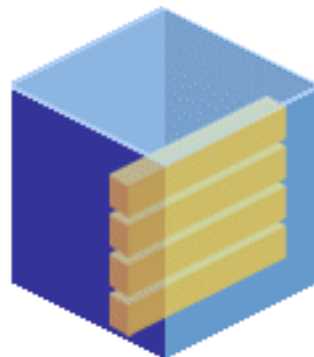
After completing this lesson, you should be able to do the following:

- Distinguish between implicit and explicit cursors
- Discuss the reasons for using explicit cursors
- Declare and control explicit cursors
- Use simple loops and cursor `FOR` loops to fetch data
- Declare and use cursors with parameters
- Lock rows with the `FOR UPDATE` clause
- Reference the current row with the `WHERE CURRENT OF` clause

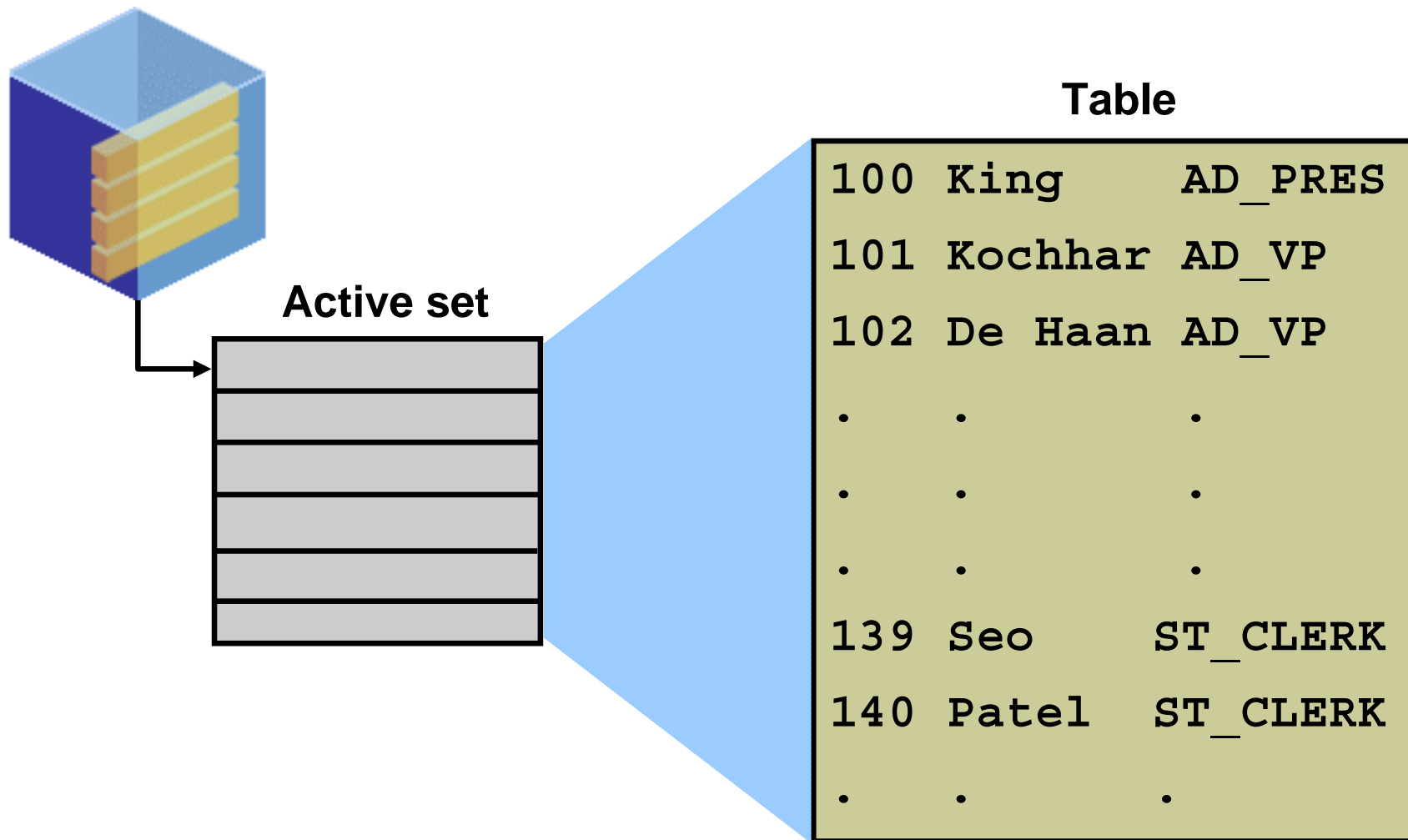
# Cursors

Every SQL statement executed by the Oracle server has an associated individual cursor:

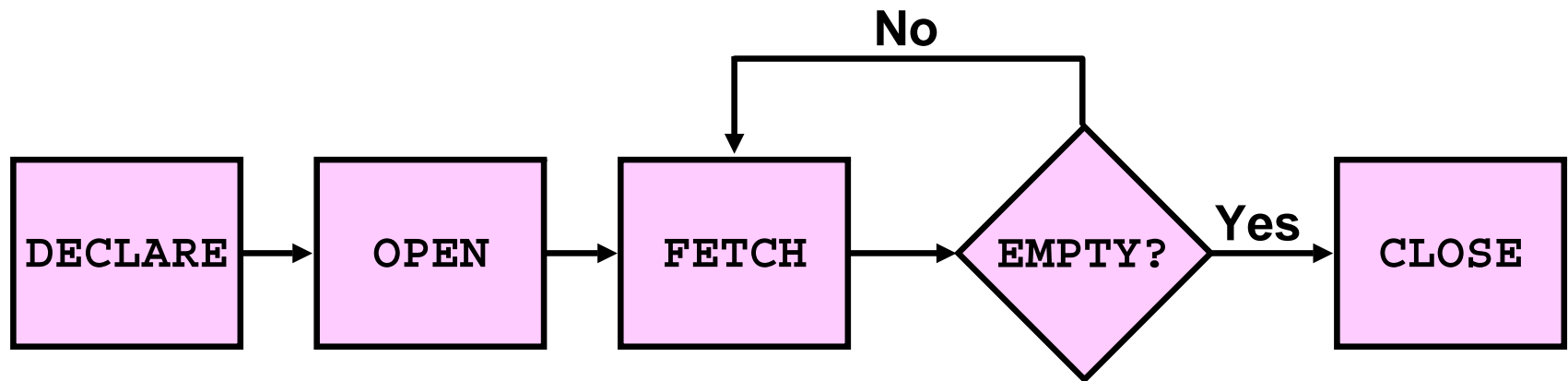
- Implicit cursors: Declared and managed by PL/SQL for all DML and PL/SQL SELECT statements
- Explicit cursors: Declared and managed by the programmer



# Explicit Cursor Operations



# Controlling Explicit Cursors

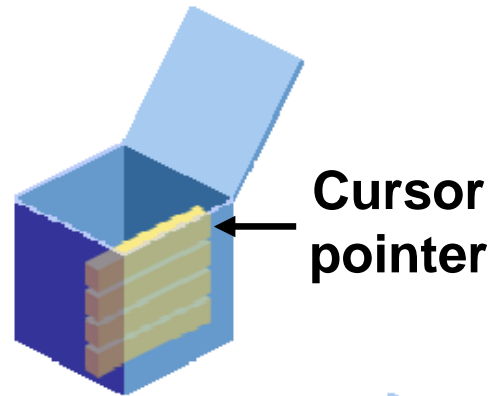


- Create a named SQL area.
- Identify the active set.
- Load the current row into variables.
- Test for existing rows.
  - Return to FETCH if rows are found.
- Release the active set.

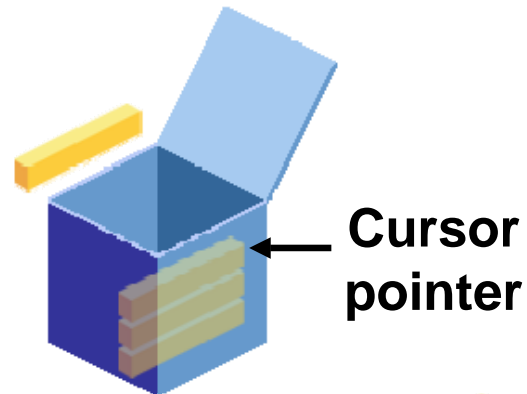


# Controlling Explicit Cursors

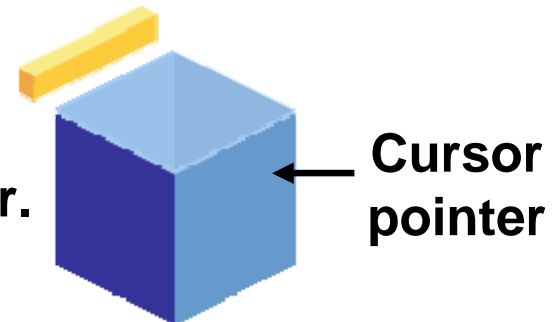
**1** Open the cursor.



**2** Fetch a row.



**3** Close the cursor.



# Declaring the Cursor

Syntax:

```
CURSOR cursor_name IS  
    select_statement;
```

Examples:

```
DECLARE  
    CURSOR emp_cursor IS  
        SELECT employee_id, last_name FROM employees  
        WHERE department_id =30;
```

```
DECLARE  
    locid NUMBER:= 1700;  
    CURSOR dept_cursor IS  
        SELECT * FROM departments  
        WHERE location_id = locid;  
    ...
```

# Opening the Cursor

```
DECLARE
  CURSOR emp_cursor IS
    SELECT employee_id, last_name FROM employees
    WHERE department_id =30;
  ...
BEGIN
  OPEN emp_cursor;
```

# Fetching Data from the Cursor

```
SET SERVEROUTPUT ON
DECLARE
  CURSOR emp_cursor IS
    SELECT employee_id, last_name FROM employees
    WHERE department_id =30;
  empno employees.employee_id%TYPE;
  lname employees.last_name%TYPE;
BEGIN
  OPEN emp_cursor;
  FETCH emp_cursor INTO empno, lname;
  DBMS_OUTPUT.PUT_LINE( empno || ' ' || lname);
  ...
END;
/
```

# Fetching Data from the Cursor

```
SET SERVEROUTPUT ON
DECLARE
    CURSOR emp_cursor IS
        SELECT employee_id, last_name FROM employees
        WHERE department_id =30;
    empno employees.employee_id%TYPE;
    lname employees.last_name%TYPE;
BEGIN
    OPEN emp_cursor;
    LOOP
        FETCH emp_cursor INTO empno, lname;
        EXIT WHEN emp_cursor%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE( empno || ' ' || lname);
    END LOOP;
    ...
END;
/
```

# Closing the Cursor

```
...  
  LOOP  
    FETCH emp_cursor INTO empno, lname;  
    EXIT WHEN emp_cursor%NOTFOUND;  
    DBMS_OUTPUT.PUT_LINE( empno || ' ' || lname);  
  END LOOP;  
  CLOSE emp_cursor;  
END;  
/
```

# Cursors and Records

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

```
DECLARE
  CURSOR emp_cursor IS
    SELECT employee_id, last_name FROM employees
    WHERE department_id =30;
  emp_record emp_cursor%ROWTYPE;
BEGIN
  OPEN emp_cursor;
  LOOP
    FETCH emp_cursor INTO emp_record;
    ...
```

# Cursor FOR Loops

Syntax:

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

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



# Cursor FOR Loops

```
SET SERVEROUTPUT ON
DECLARE
  CURSOR emp_cursor IS
    SELECT employee_id, last_name FROM employees
    WHERE department_id =30;
BEGIN
  FOR emp_record IN emp_cursor
  LOOP
    DBMS_OUTPUT.PUT_LINE( emp_record.employee_id
      || ' ' || emp_record.last_name );
  END LOOP;
END;
/
```

# Explicit Cursor Attributes

Obtain status information about a cursor.

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

## **%ISOPEN Attribute**

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

Example:

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

# %ROWCOUNT and %NOTFOUND: Example

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

# Cursor FOR Loops Using Subqueries

There is no need to declare the cursor.

Example:

```
SET SERVEROUTPUT ON
BEGIN
  FOR emp_record IN (SELECT employee_id, last_name
                     FROM employees WHERE department_id =30)
  LOOP
    DBMS_OUTPUT.PUT_LINE( emp_record.employee_id || '
    ' || emp_record.last_name);
  END LOOP;
END;
/
```

# Cursors with Parameters

Syntax:

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

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

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

# Cursors with Parameters

```
SET SERVEROUTPUT ON
DECLARE
    CURSOR    emp_cursor (deptno NUMBER) IS
        SELECT employee_id, last_name
        FROM    employees
        WHERE   department_id = deptno;
    dept_id NUMBER;
    lname     VARCHAR2(15);
BEGIN
    OPEN emp_cursor (10);
    ...
    CLOSE emp_cursor;
    OPEN emp_cursor (20);
    ...
```

# FOR UPDATE Clause

Syntax:

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

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



# WHERE CURRENT OF Clause

Syntax:

```
WHERE CURRENT OF cursor ;
```

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

```
UPDATE employees  
  SET      salary = ...  
  WHERE CURRENT OF emp_cursor;
```

# Cursors with Subqueries

Example:

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

```

# Summary

In this lesson, you should have learned how to:

- Distinguish cursor types:
  - Implicit cursors are used for all `DML` statements and single-row queries.
  - Explicit cursors are used for queries of zero, one, or more rows.
- Create and handle explicit cursors
- Use simple loops and cursor `FOR` loops to handle multiple rows in the cursors
- Evaluate the cursor status by using the cursor attributes
- Use the `FOR UPDATE` and `WHERE CURRENT OF` clauses to update or delete the current fetched row

# Practice 7: Overview

This practice covers the following topics:

- Declaring and using explicit cursors to query rows of a table
- Using a cursor `FOR` loop
- Applying cursor attributes to test the cursor status
- Declaring and using cursors with parameters
- Using the `FOR UPDATE` and `WHERE CURRENT OF` clauses

# 8

## Handling Exceptions

# Objectives

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

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

# Example of an Exception

```
SET SERVEROUTPUT ON
DECLARE
    lname VARCHAR2(15);
BEGIN
    SELECT last_name INTO lname FROM employees WHERE
    first_name='John';
    DBMS_OUTPUT.PUT_LINE ('John''s last name is : '
    ||lname);
END;
/
```

# Example of an Exception

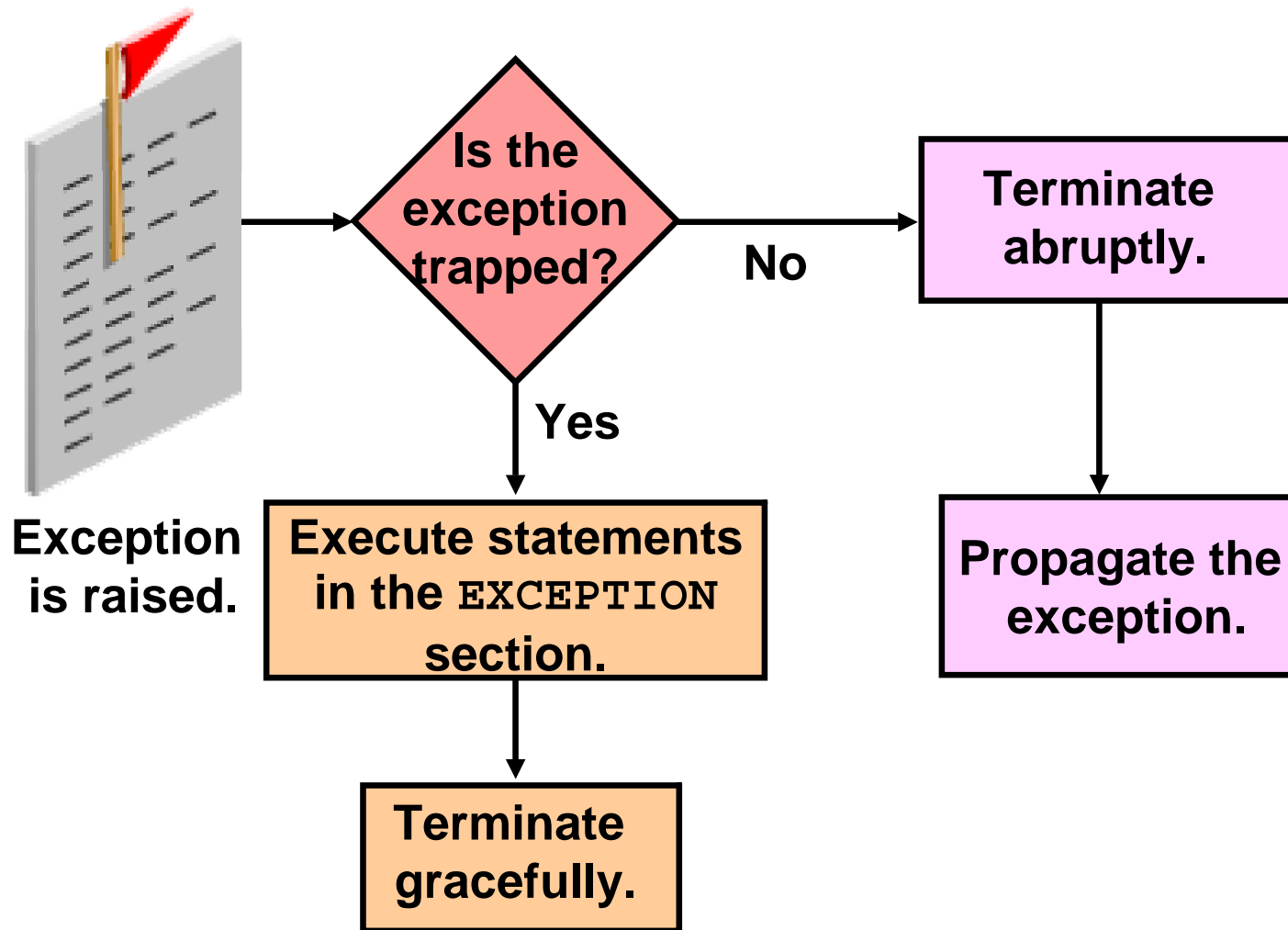
```
SET SERVEROUTPUT ON
DECLARE
    lname VARCHAR2(15);
BEGIN
    SELECT last_name INTO lname FROM employees WHERE
    first_name='John';
    DBMS_OUTPUT.PUT_LINE ('John's last name is : '
    ||lname);
EXCEPTION
    WHEN TOO_MANY_ROWS THEN
    DBMS_OUTPUT.PUT_LINE (' Your select statement
    retrieved multiple rows. Consider using a
    cursor. ');
END;
/
```




# Handling Exceptions with PL/SQL

- An exception is a PL/SQL error that is raised during program execution.
- An exception can be raised:
  - Implicitly by the Oracle server
  - Explicitly by the program
- An exception can be handled:
  - By trapping it with a handler
  - By propagating it to the calling environment

# Handling Exceptions



# Exception Types

- Predefined Oracle server
  - Non-predefined Oracle server
- 
- Implicitly raised
- 
- User-defined
- Explicitly raised

# Trapping Exceptions

Syntax:

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

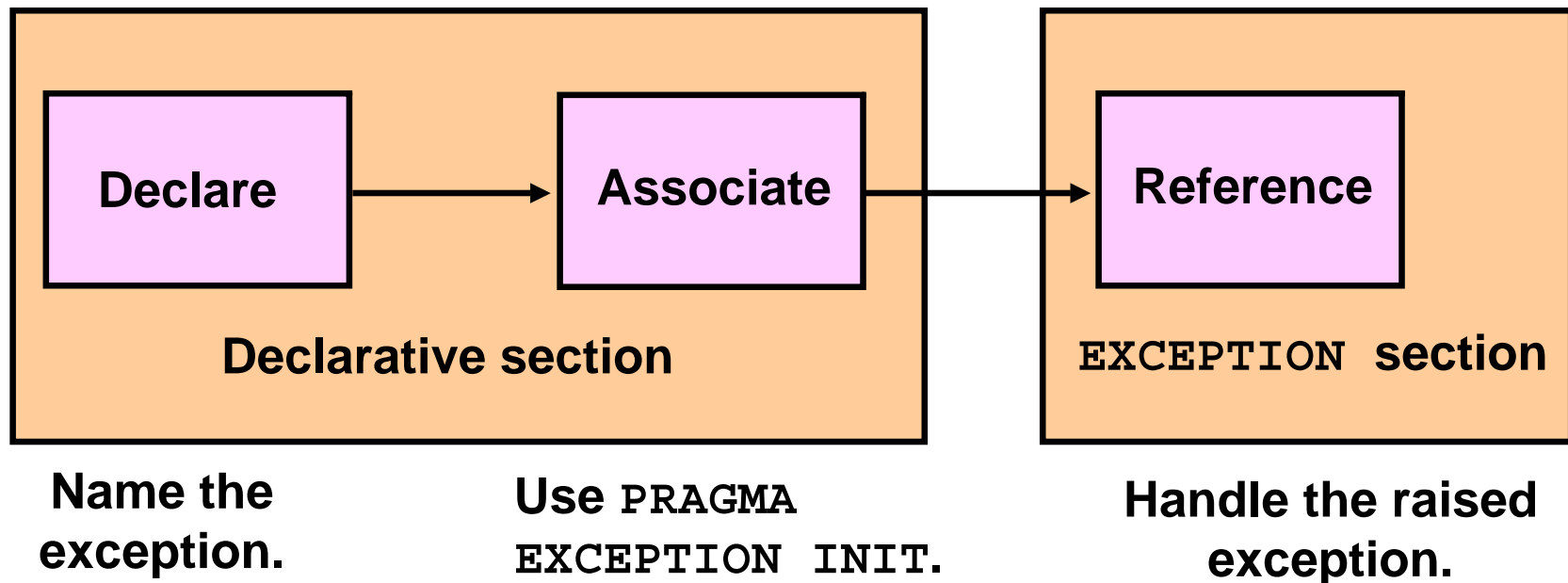
# Guidelines for Trapping Exceptions

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

# Trapping Predefined Oracle Server Errors

- Reference the predefined name in the exception-handling routine.
- Sample predefined exceptions:
  - NO\_DATA\_FOUND
  - TOO\_MANY\_ROWS
  - INVALID\_CURSOR
  - ZERO\_DIVIDE
  - DUP\_VAL\_ON\_INDEX

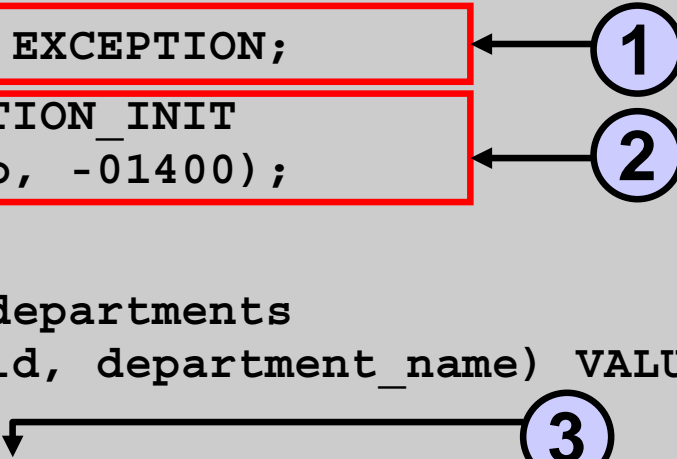
# Trapping Non-Predefined Oracle Server Errors



# Non-Predefined Error

To trap Oracle server error number -01400  
("cannot insert NULL"):

```
SET SERVEROUTPUT ON
DECLARE
  insert_excep EXCEPTION;
  PRAGMA EXCEPTION_INIT
    (insert_excep, -01400);
BEGIN
  INSERT INTO departments
    (department_id, department_name) VALUES (280, NULL);
EXCEPTION
  WHEN insert_excep THEN
    DBMS_OUTPUT.PUT_LINE('INSERT OPERATION FAILED');
    DBMS_OUTPUT.PUT_LINE(SQLERRM);
END;
/
```





# Functions for Trapping Exceptions

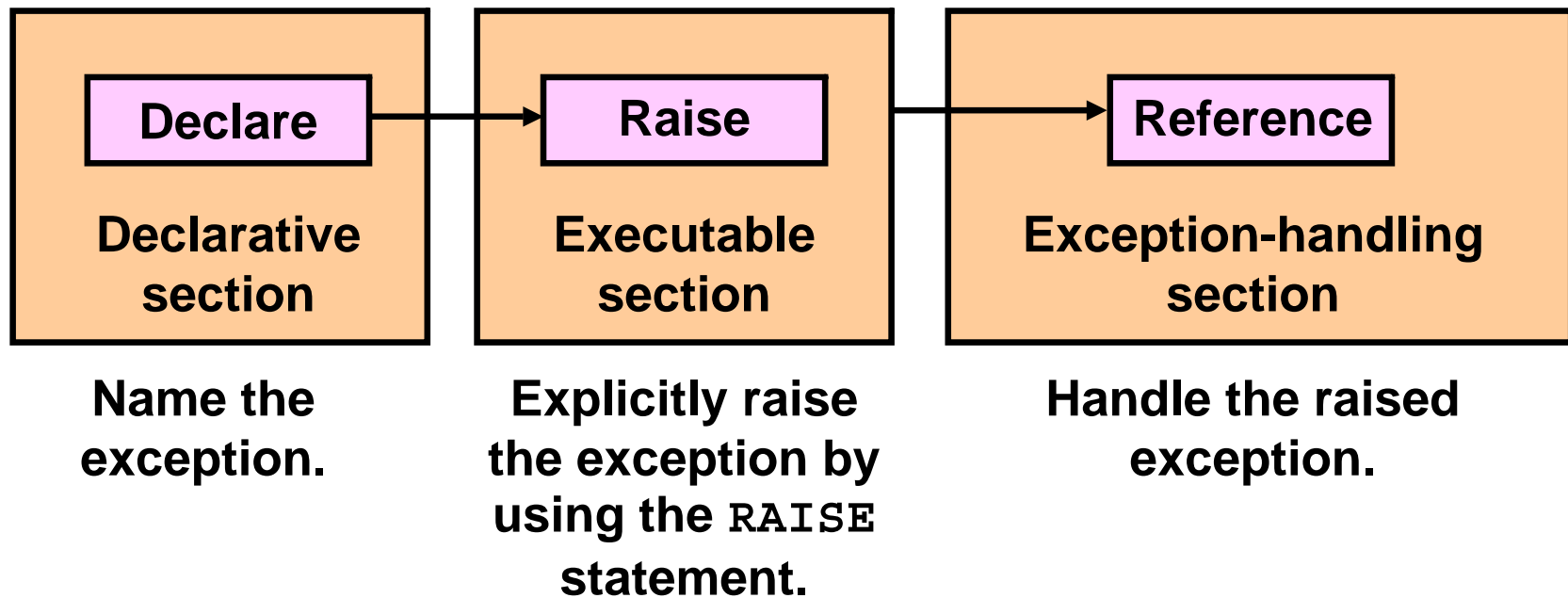
- `SQLCODE`: Returns the numeric value for the error code
- `SQLERRM`: Returns the message associated with the error number

# Functions for Trapping Exceptions

Example:

```
DECLARE
    error_code      NUMBER;
    error_message    VARCHAR2(255);
BEGIN
    ...
EXCEPTION
    ...
    WHEN OTHERS THEN
        ROLLBACK;
        error_code := SQLCODE;
        error_message := SQLERRM;
        INSERT INTO errors (e_user, e_date, error_code,
            error_message) VALUES (USER, SYSDATE, error_code,
            error_message);
END;
/
```

# Trapping User-Defined Exceptions



# Trapping User-Defined Exceptions

```
ACCEPT deptno PROMPT 'Please enter the department number:'
ACCEPT name    PROMPT 'Please enter the department name:'
DECLARE
  invalid_department EXCEPTION; ← ①
  name VARCHAR2(20) := '&name';
  deptno NUMBER := &deptno;
BEGIN
  UPDATE departments
  SET    department_name = name
  WHERE  department_id = deptno;
  IF SQL%NOTFOUND THEN
    RAISE invalid_department; ← ②
  END IF;
  COMMIT;
EXCEPTION
  WHEN invalid_department THEN ← ③
    DBMS_OUTPUT.PUT_LINE('No such department id.');
```

END;

/

# Calling Environments

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

# Propagating Exceptions in a Subblock

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

```
DECLARE
    . . .
    no_rows          exception;
    integrity        exception;
    PRAGMA EXCEPTION_INIT (integrity, -2292);
BEGIN
    FOR c_record IN emp_cursor LOOP
        BEGIN
            SELECT ...
            UPDATE ...
            IF SQL%NOTFOUND THEN
                RAISE no_rows;
            END IF;
        END;
    END LOOP;
EXCEPTION
    WHEN integrity THEN ...
    WHEN no_rows THEN ...
END;
/
```

# RAISE\_APPLICATION\_ERROR Procedure

Syntax:

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

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

# **RAISE\_APPLICATION\_ERROR Procedure**

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



# RAISE\_APPLICATION\_ERROR Procedure

Executable section:

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

Exception section:

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

END;  
/

# Summary

In this lesson, you should have learned how to:

- Define PL/SQL exceptions
- Add an `EXCEPTION` section to the PL/SQL block to deal with exceptions at run time
- Handle different types of exceptions:
  - Predefined exceptions
  - Non-predefined exceptions
  - User-defined exceptions
- Propagate exceptions in nested blocks and call applications

# Practice 8: Overview

This practice covers the following topics:

- Handling named exceptions
- Creating and invoking user-defined exceptions



# Creating Stored Procedures and Functions

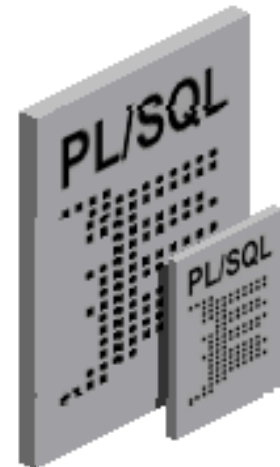
# Objectives

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

- Differentiate between anonymous blocks and subprograms
- Create a simple procedure and invoke it from an anonymous block
- Create a simple function
- Create a simple function that accepts a parameter
- Differentiate between procedures and functions

# Procedures and Functions

- Are named PL/SQL blocks
- Are called PL/SQL subprograms
- Have block structures similar to anonymous blocks:
  - Optional declarative section (without `DECLARE` keyword)
  - Mandatory executable section
  - Optional section to handle exceptions



# Differences Between Anonymous Blocks and Subprograms

Anonymous Blocks	Subprograms
Unnamed PL/SQL blocks	Named PL/SQL blocks
Compiled every time	Compiled only once
Not stored in the database	Stored in the database
Cannot be invoked by other applications	Named and, therefore, can be invoked by other applications
Do not return values	Subprograms called functions must return values.
Cannot take parameters	Can take parameters

# Procedure: Syntax

```
CREATE [OR REPLACE] PROCEDURE procedure_name
  [(argument1 [mode1] datatype1,
    argument2 [mode2] datatype2,
    . . .)]
IS|AS
procedure_body;
```



# Procedure: Example

```
CREATE TABLE dept AS SELECT * FROM departments;
CREATE PROCEDURE add_dept IS
    dept_id dept.department_id%TYPE;
    dept_name dept.department_name%TYPE;
BEGIN
    dept_id:=280;
    dept_name:='ST-Curriculum';
    INSERT INTO dept(department_id,department_name)
    VALUES(dept_id,dept_name);
    DBMS_OUTPUT.PUT_LINE(' Inserted ' ||
        SQL%ROWCOUNT || ' row ');
END;
/
```

# Invoking the Procedure

```
BEGIN
  add_dept;
END;
/
SELECT department_id, department_name FROM dept
WHERE department_id=280;
```

anonymous block completed  
Inserted 1 row

	DEPARTMENT_ID	DEPARTMENT_NAME
1	280	ST-Curriculum

# Function: Syntax

```
CREATE [OR REPLACE] FUNCTION function_name
  [(argument1 [mode1] datatype1,
    argument2 [mode2] datatype2,
    . . .)]
RETURN datatype
IS|AS
function_body;
```

# Function: Example

```
CREATE FUNCTION check_sal RETURN Boolean IS
  dept_id employees.department_id%TYPE;
  empno    employees.employee_id%TYPE;
  sal      employees.salary%TYPE;
  avg_sal  employees.salary%TYPE;
BEGIN
  empno:=205;
  SELECT salary,department_id INTO sal,dept_id
  FROM employees WHERE employee_id= empno;
  SELECT avg(salary) INTO avg_sal FROM employees
  WHERE department_id=dept_id;
  IF sal > avg_sal THEN
    RETURN TRUE;
  ELSE
    RETURN FALSE;
  END IF;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    RETURN NULL;
END;
/
```

# Invoking the Function

```
SET SERVEROUTPUT ON
BEGIN
  IF (check_sal IS NULL) THEN
    DBMS_OUTPUT.PUT_LINE('The function returned
      NULL due to exception');
  ELSIF (check_sal) THEN
    DBMS_OUTPUT.PUT_LINE('Salary > average');
  ELSE
    DBMS_OUTPUT.PUT_LINE('Salary < average');
  END IF;
END;
/
```

```
anonymous block completed
Salary > average
```

# Passing a Parameter to the Function

```
DROP FUNCTION check_sal;
CREATE FUNCTION check_sal(empno employees.employee_id%TYPE)
RETURN Boolean IS
    dept_id employees.department_id%TYPE;
    sal      employees.salary%TYPE;
    avg_sal  employees.salary%TYPE;
BEGIN
    SELECT salary,department_id INTO sal,dept_id
    FROM employees WHERE employee_id=empno;
    SELECT avg(salary) INTO avg_sal FROM employees
    WHERE department_id=dept_id;
    IF sal > avg_sal THEN
        RETURN TRUE;
    ELSE
        RETURN FALSE;
    END IF;
EXCEPTION ...
...
```

# Invoking the Function with a Parameter

```
BEGIN
DBMS_OUTPUT.PUT_LINE('Checking for employee with id 205');
  IF (check_sal(205) IS NULL) THEN
    DBMS_OUTPUT.PUT_LINE('The function returned
      NULL due to exception');
  ELSIF (check_sal(205)) THEN
    DBMS_OUTPUT.PUT_LINE('Salary > average');
  ELSE
    DBMS_OUTPUT.PUT_LINE('Salary < average');
  END IF;
DBMS_OUTPUT.PUT_LINE('Checking for employee with id 70');
  IF (check_sal(70) IS NULL) THEN
    DBMS_OUTPUT.PUT_LINE('The function returned
      NULL due to exception');
  ELSIF (check_sal(70)) THEN
    ...
  END IF;
END;
/
```

# Summary

In this lesson, you should have learned how to:

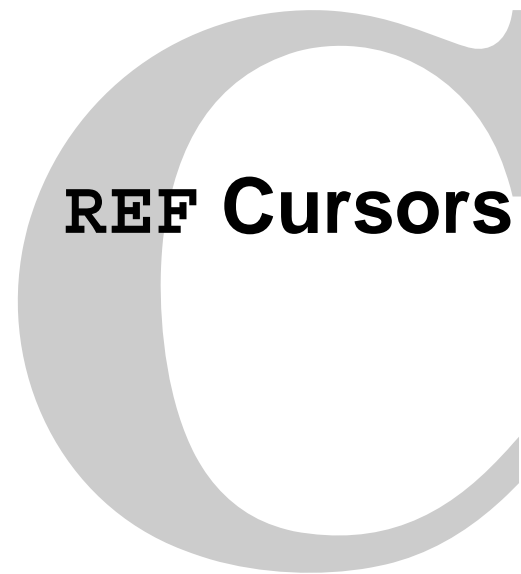
- Create a simple procedure
- Invoke the procedure from an anonymous block
- Create a simple function
- Create a simple function that accepts parameters
- Invoke the function from an anonymous block



# Practice 9: Overview

This practice covers the following topics:

- Converting an existing anonymous block to a procedure
- Modifying the procedure to accept a parameter
- Writing an anonymous block to invoke the procedure



# Cursor Variables

- Cursor variables are like C or Pascal pointers, which hold the memory location (address) of an item instead of the item itself.
- In PL/SQL, a pointer is declared as `REF X`, where `REF` is short for `REFERENCE` and `X` stands for a class of objects.
- A cursor variable has the data type `REF CURSOR`.
- A cursor is static, but a cursor variable is dynamic.
- Cursor variables give you more flexibility.

# Why Use Cursor Variables?

- You can use cursor variables to pass query result sets between PL/SQL stored subprograms and various clients.
- PL/SQL can share a pointer to the query work area in which the result set is stored.
- You can pass the value of a cursor variable freely from one scope to another.
- You can reduce network traffic by having a PL/SQL block open (or close) several host cursor variables in a single round trip.

# Defining REF CURSOR Types

- Define a REF CURSOR type:

```
Define a REF CURSOR type  
TYPE ref_type_name IS REF CURSOR [RETURN  
return_type];
```

- Declare a cursor variable of that type:

```
ref_cv ref_type_name;
```

- Example:

```
DECLARE  
TYPE DeptCurTyp IS REF CURSOR RETURN  
departments%ROWTYPE;  
dept_cv DeptCurTyp;
```

# Using the OPEN-FOR, FETCH, and CLOSE Statements

- The OPEN-FOR statement associates a cursor variable with a multirow query, executes the query, identifies the result set, and positions the cursor to point to the first row of the result set.
- The FETCH statement returns a row from the result set of a multirow query, assigns the values of select-list items to corresponding variables or fields in the INTO clause, increments the count kept by %ROWCOUNT, and advances the cursor to the next row.
- The CLOSE statement disables a cursor variable.

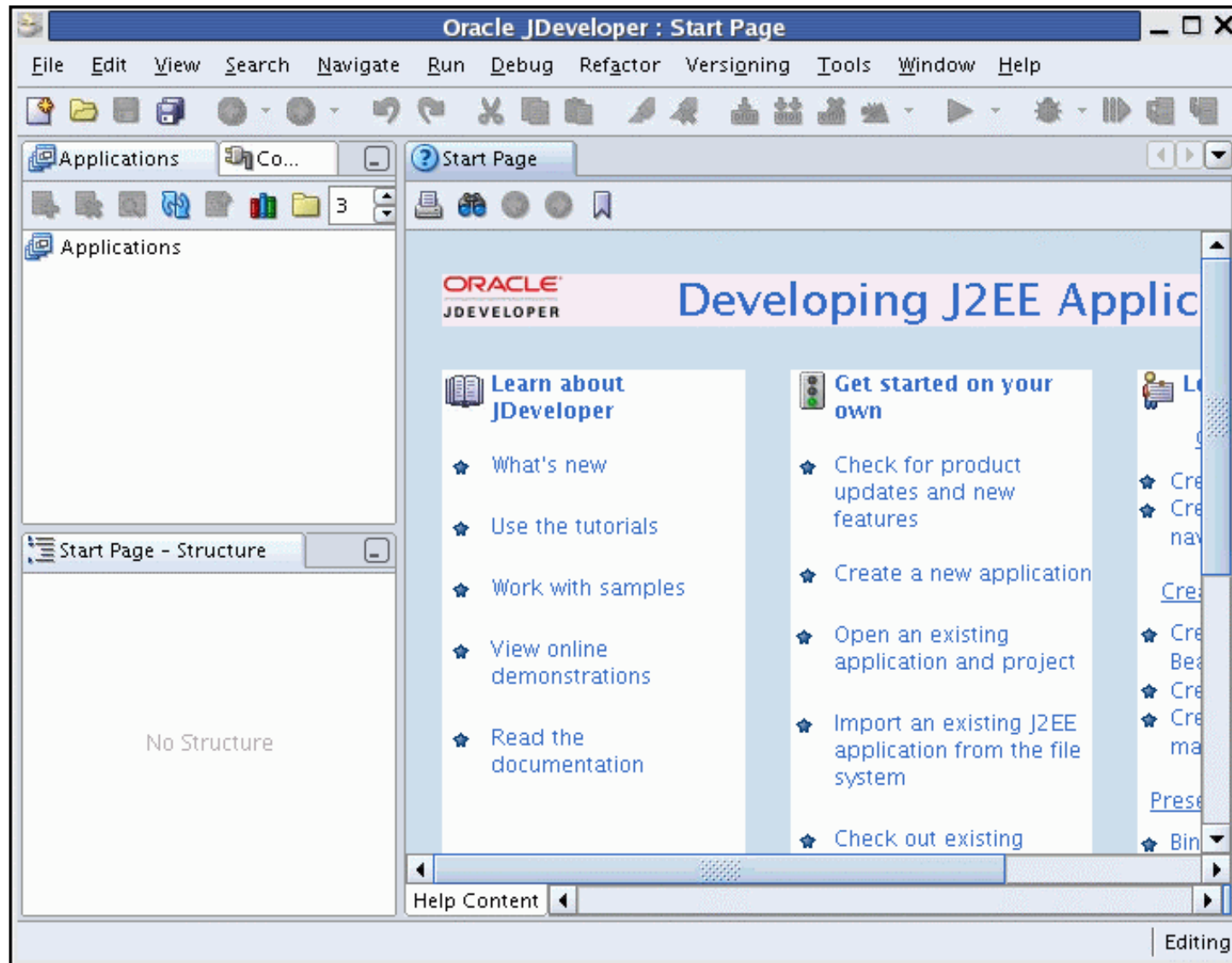
# An Example of Fetching

```
DECLARE
    TYPE EmpCurTyp IS REF CURSOR;
    emp_cv      EmpCurTyp;
    emp_rec     employees%ROWTYPE;
    sql_stmt    VARCHAR2(200);
    my_job      VARCHAR2(10) := 'ST_CLERK';
BEGIN
    sql_stmt := 'SELECT * FROM employees
                WHERE job_id = :j';
    OPEN emp_cv FOR sql_stmt USING my_job;
    LOOP
        FETCH emp_cv INTO emp_rec;
        EXIT WHEN emp_cv%NOTFOUND;
        -- process record
    END LOOP;
    CLOSE emp_cv;
END;
/
```

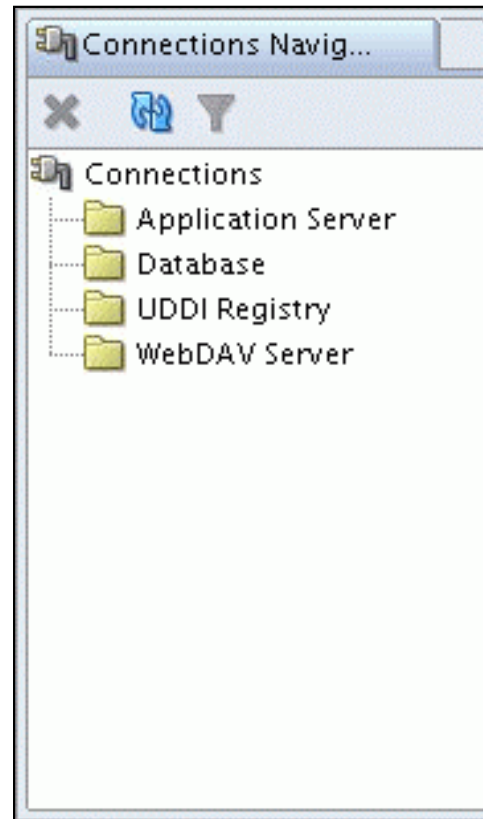




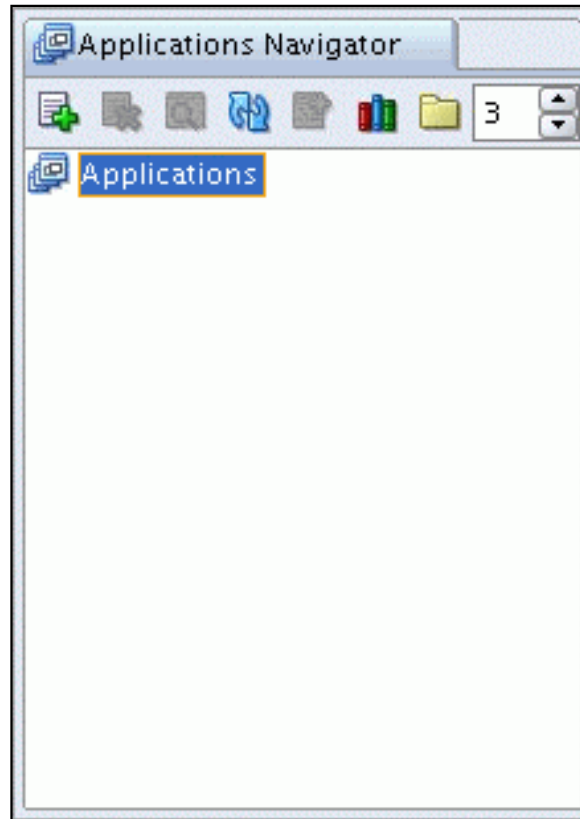
# Oracle JDeveloper 10g



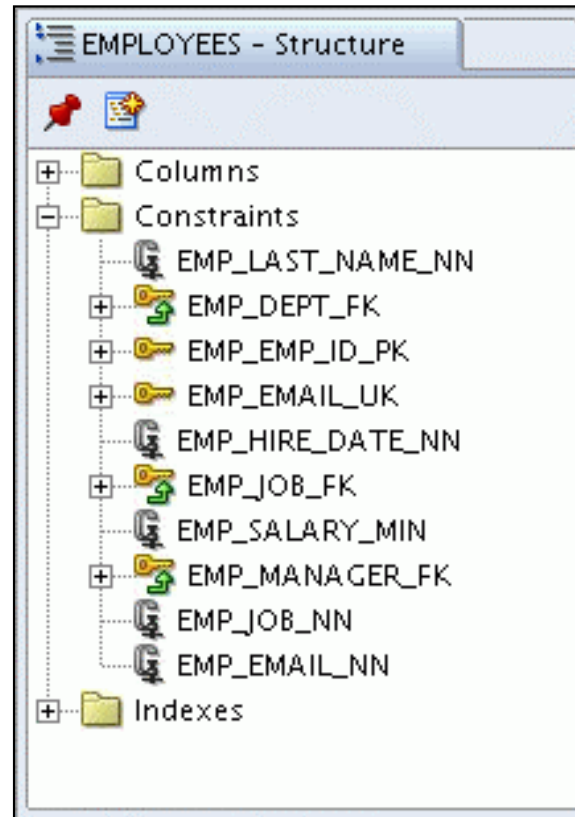
# Connection Navigator



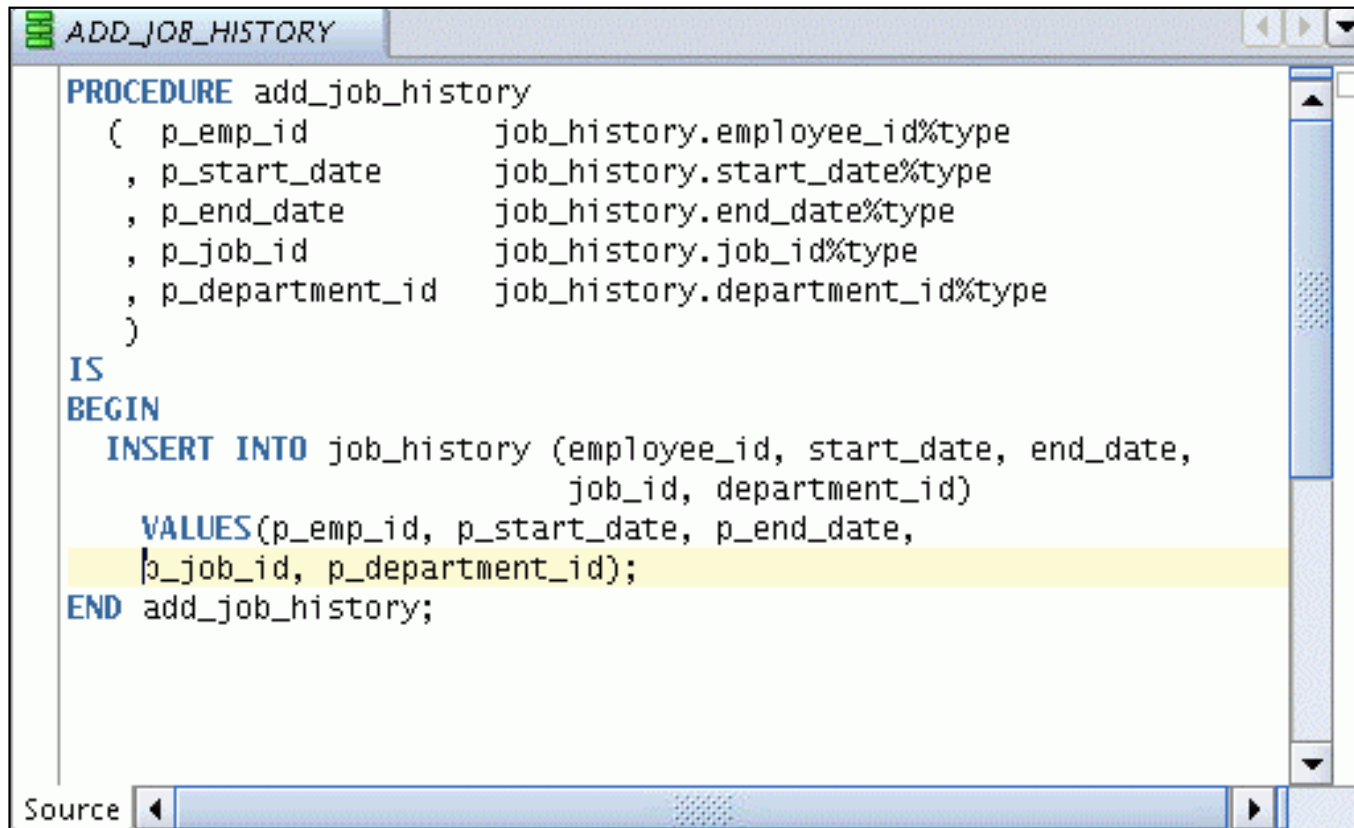
# Application Navigator



# Structure Window



# Editor Window



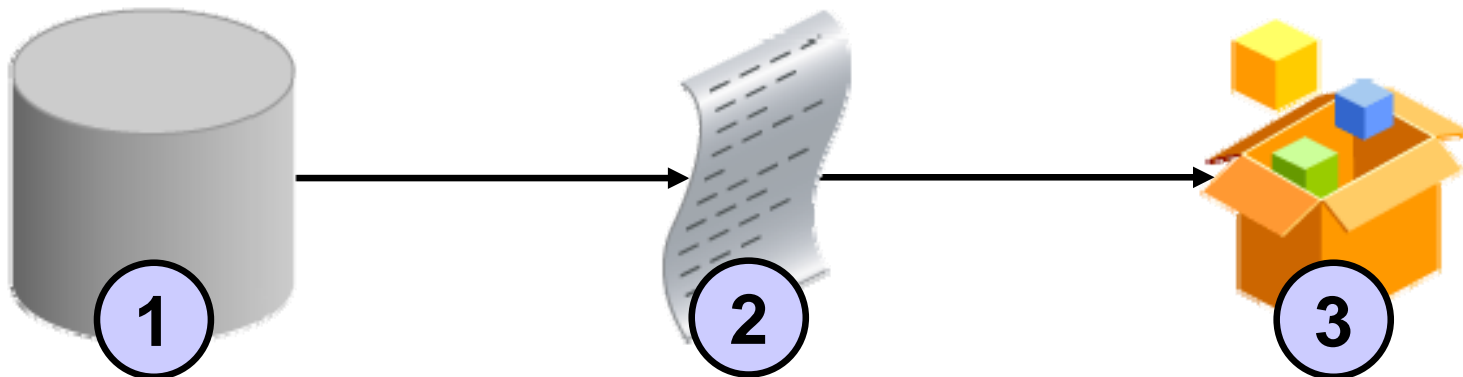
```
ADD_JOB_HISTORY

PROCEDURE add_job_history
(
  p_emp_id          job_history.employee_id%type
, p_start_date      job_history.start_date%type
, p_end_date        job_history.end_date%type
, p_job_id          job_history.job_id%type
, p_department_id   job_history.department_id%type
)
IS
BEGIN
  INSERT INTO job_history (employee_id, start_date, end_date,
                           job_id, department_id)
    VALUES(p_emp_id, p_start_date, p_end_date,
            p_job_id, p_department_id);
END add_job_history;
```

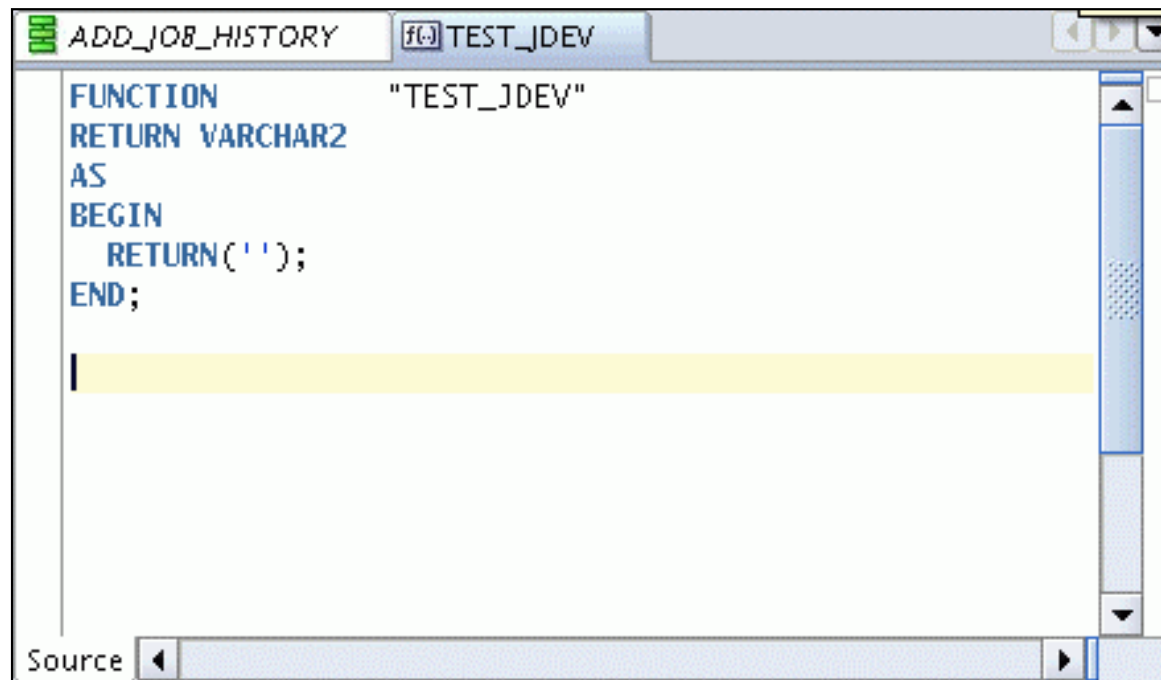
# Deploying Java Stored Procedures

Before deploying Java stored procedures, perform the following steps:

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



# Creating Program Units



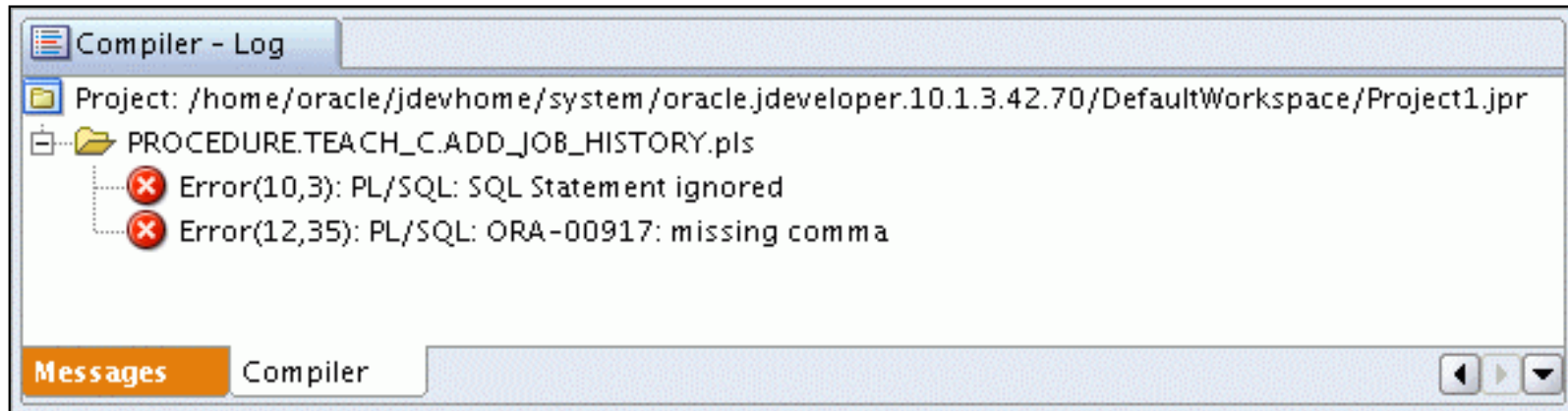
The screenshot shows a window titled 'ADD\_JOB\_HISTORY' with a sub-tab 'TEST\_JDEV'. The main area contains the following SQL code:

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

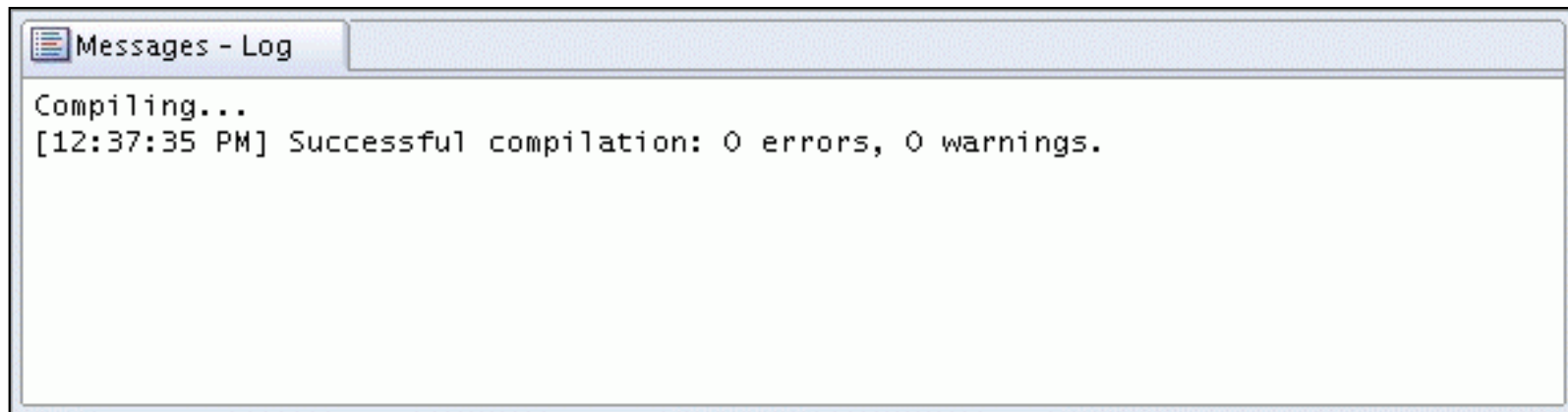
The code is displayed in a monospaced font. A yellow highlight bar is visible below the 'END;' statement. The window has a standard scrollbar on the right and a status bar at the bottom labeled 'Source'.

**Skeleton of the function**

# Compiling



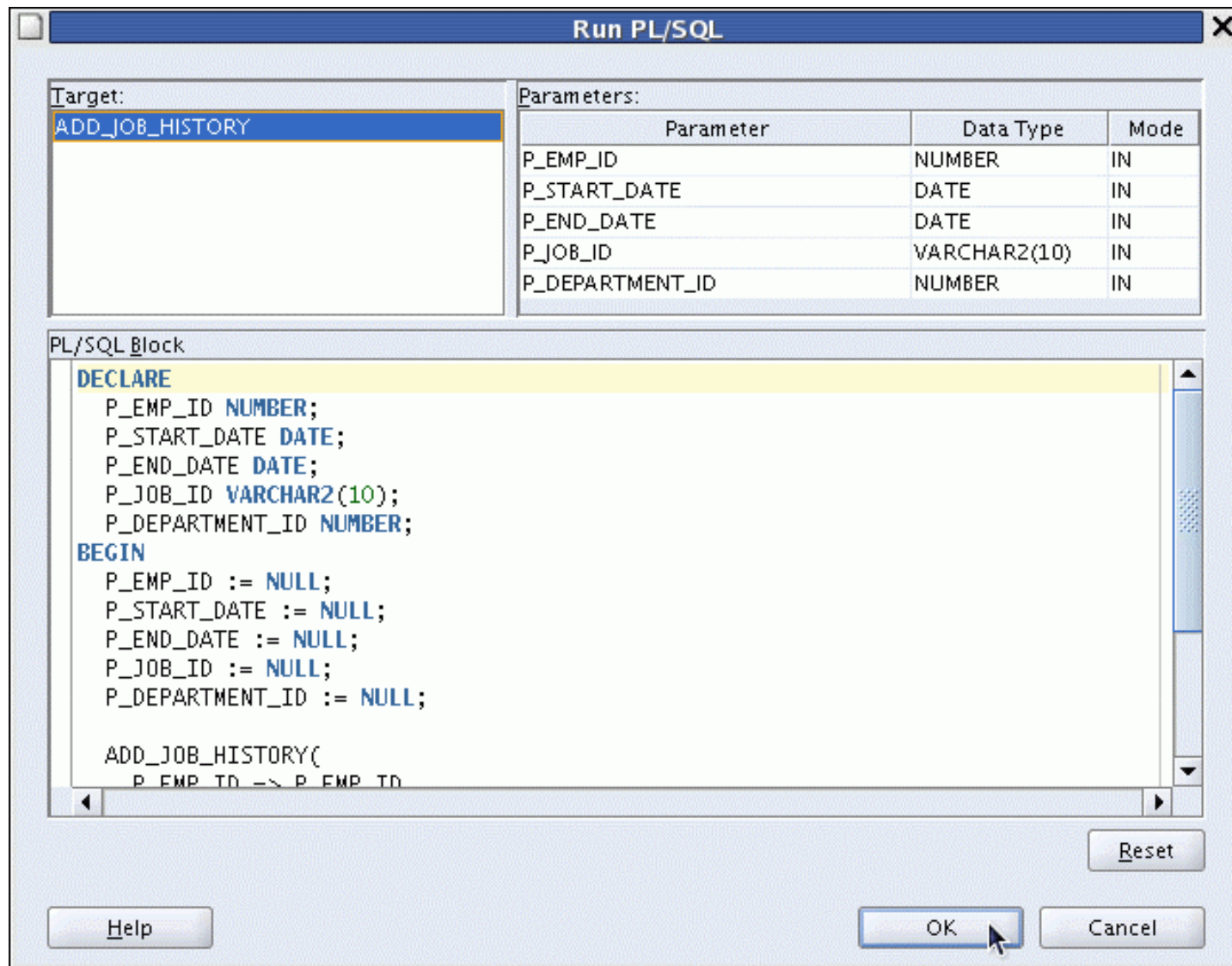
## Compilation with errors



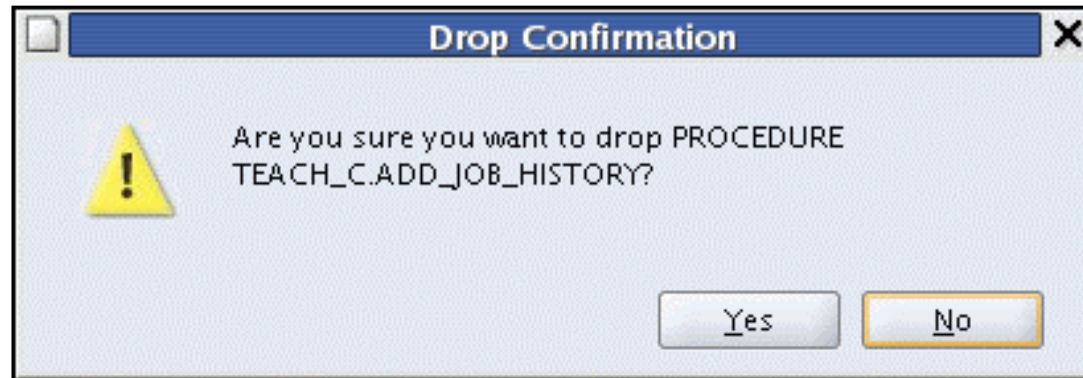
## Compilation without errors



# Running a Program Unit



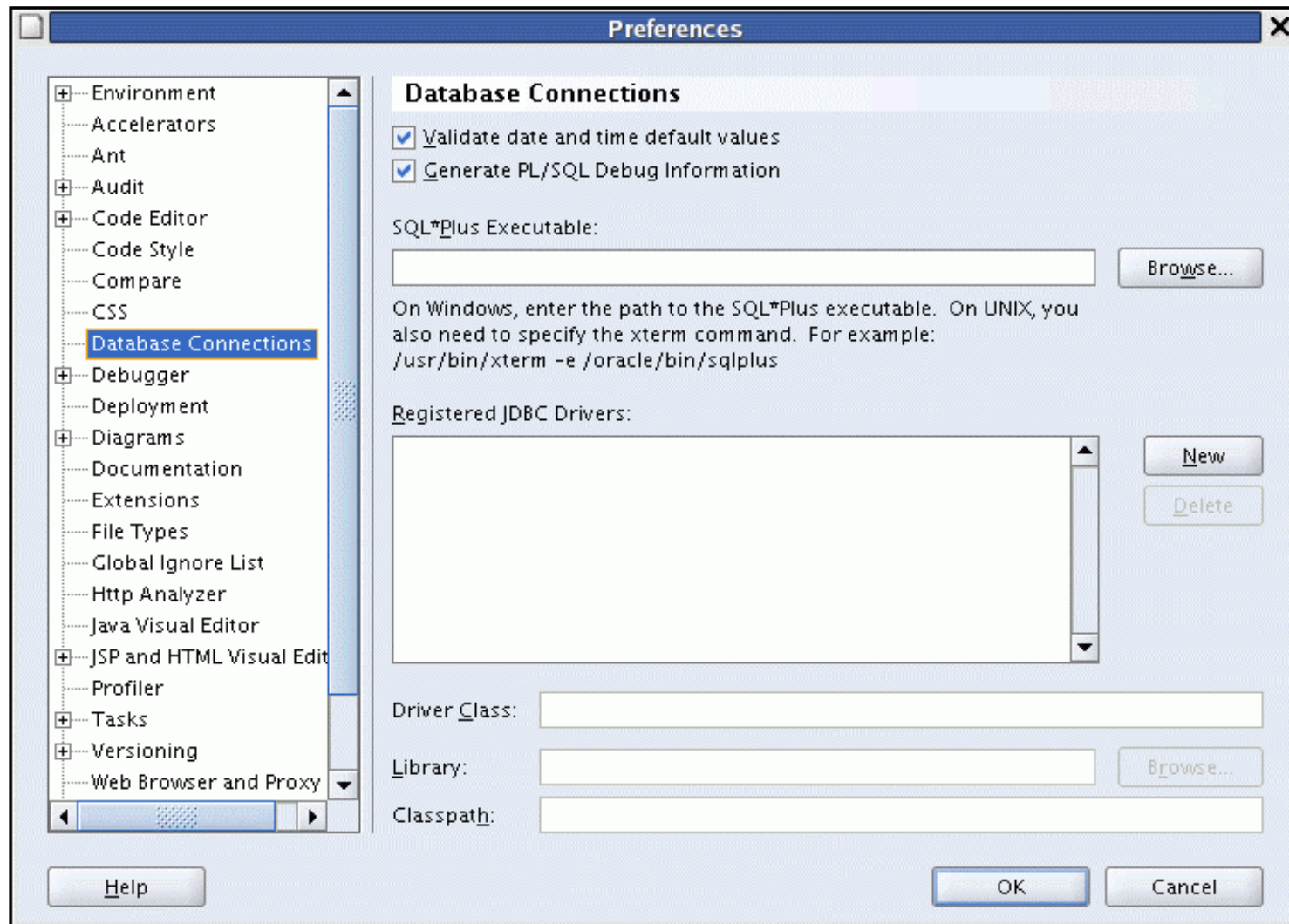
# Dropping a Program Unit



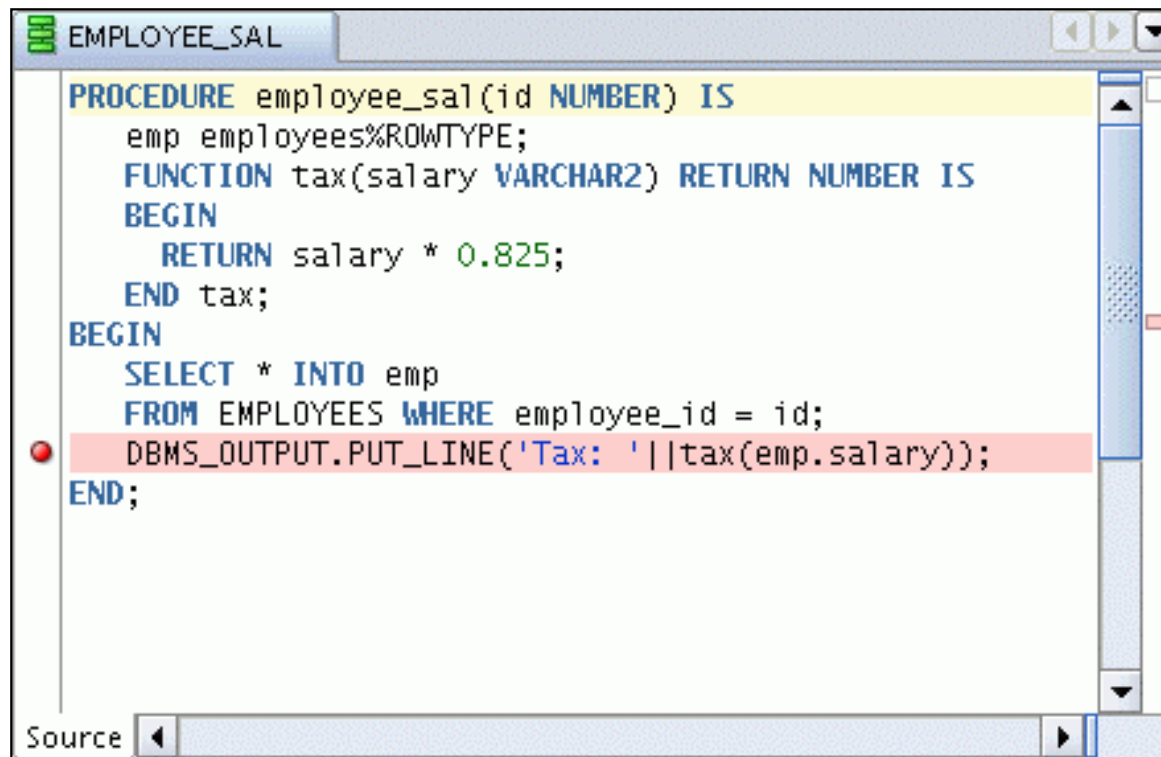
# Debugging PL/SQL Programs

- 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

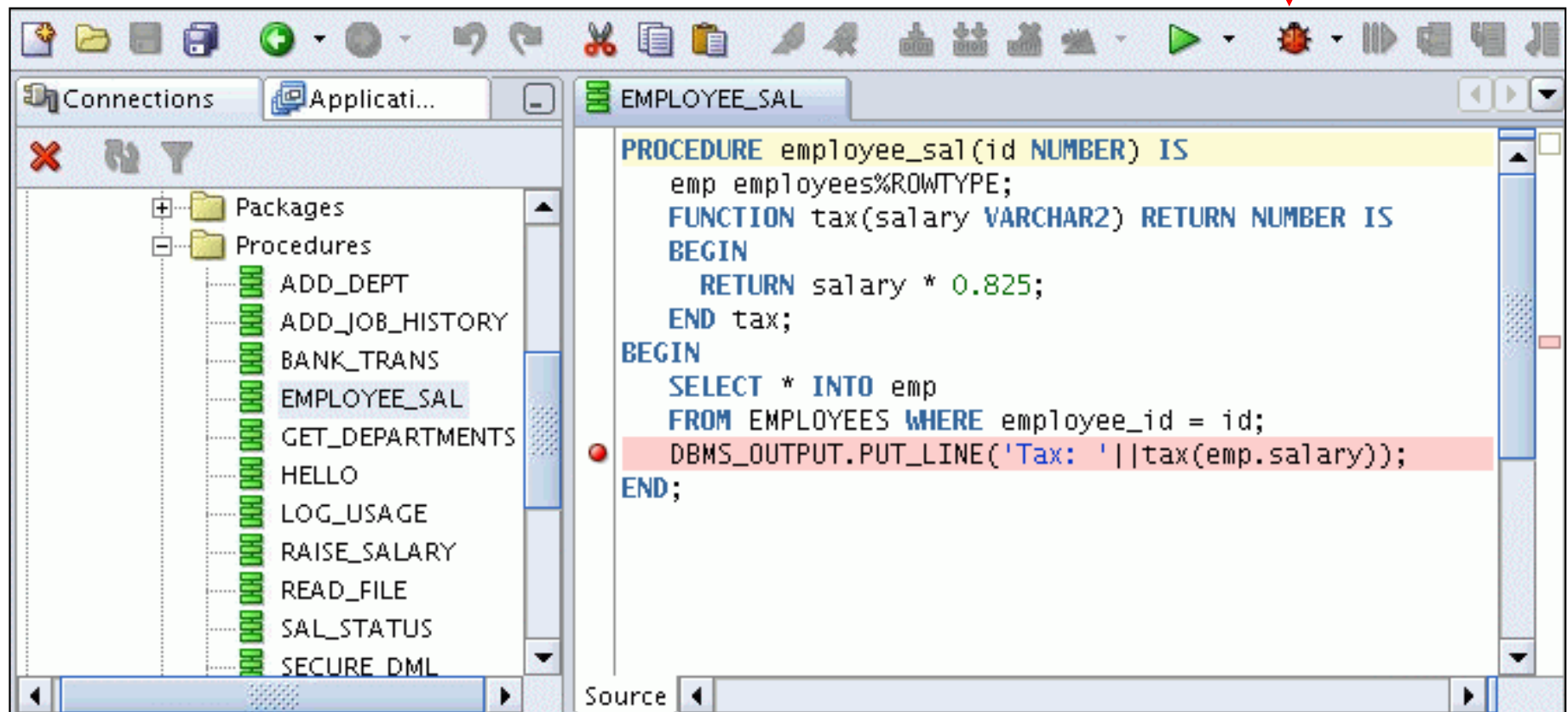


```
PROCEDURE employee_sal(id NUMBER) IS
  emp employees%ROWTYPE;
  FUNCTION tax(salary VARCHAR2) RETURN NUMBER IS
  BEGIN
    RETURN salary * 0.825;
  END tax;
BEGIN
  SELECT * INTO emp
  FROM EMPLOYEES WHERE employee_id = id;
  DBMS_OUTPUT.PUT_LINE('Tax: ' || tax(emp.salary));
END;
```

The screenshot shows a window titled 'EMPLOYEE\_SAL' containing PL/SQL code. A red circle breakpoint is placed on the line `DBMS_OUTPUT.PUT_LINE('Tax: ' || tax(emp.salary));`. The code defines a procedure `employee_sal` that takes an `id` parameter, selects an employee record, and outputs the tax calculated by a nested function `tax`.

# Stepping Through Code

Debug





# Using SQL Developer

# Objectives

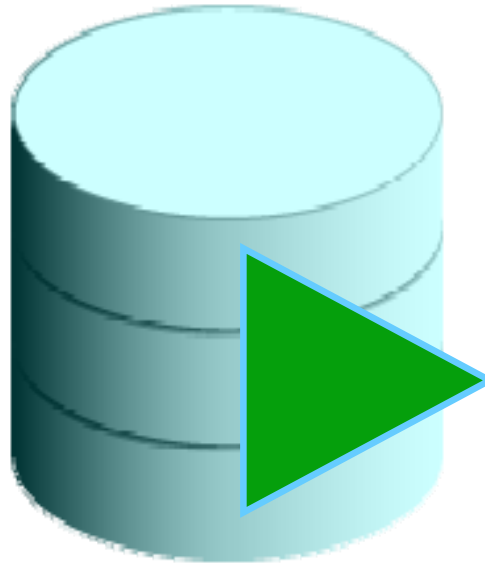
After completing this appendix, you should be able to do the following:

- 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 SQL Worksheet
- Execute SQL statements and SQL scripts
- Create and save reports



# What Is Oracle SQL Developer?

- Oracle SQL Developer is a free 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.

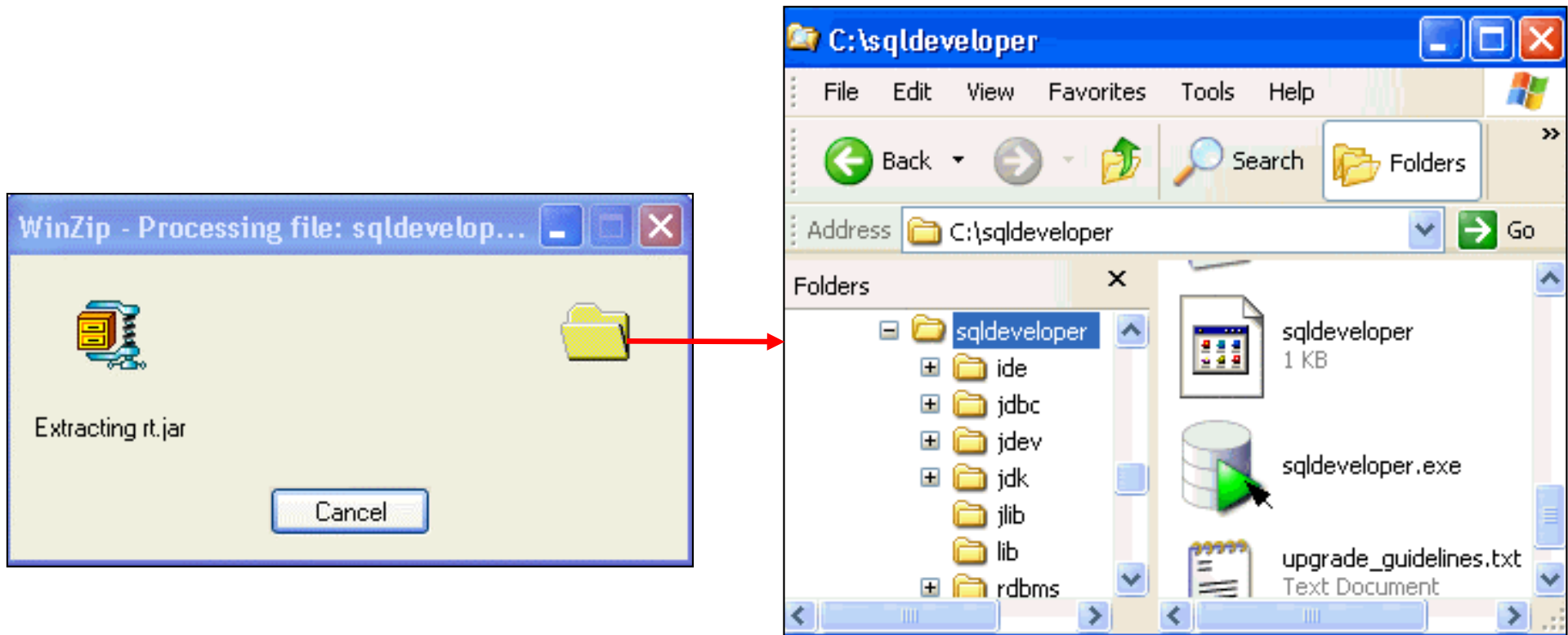


# Key Features

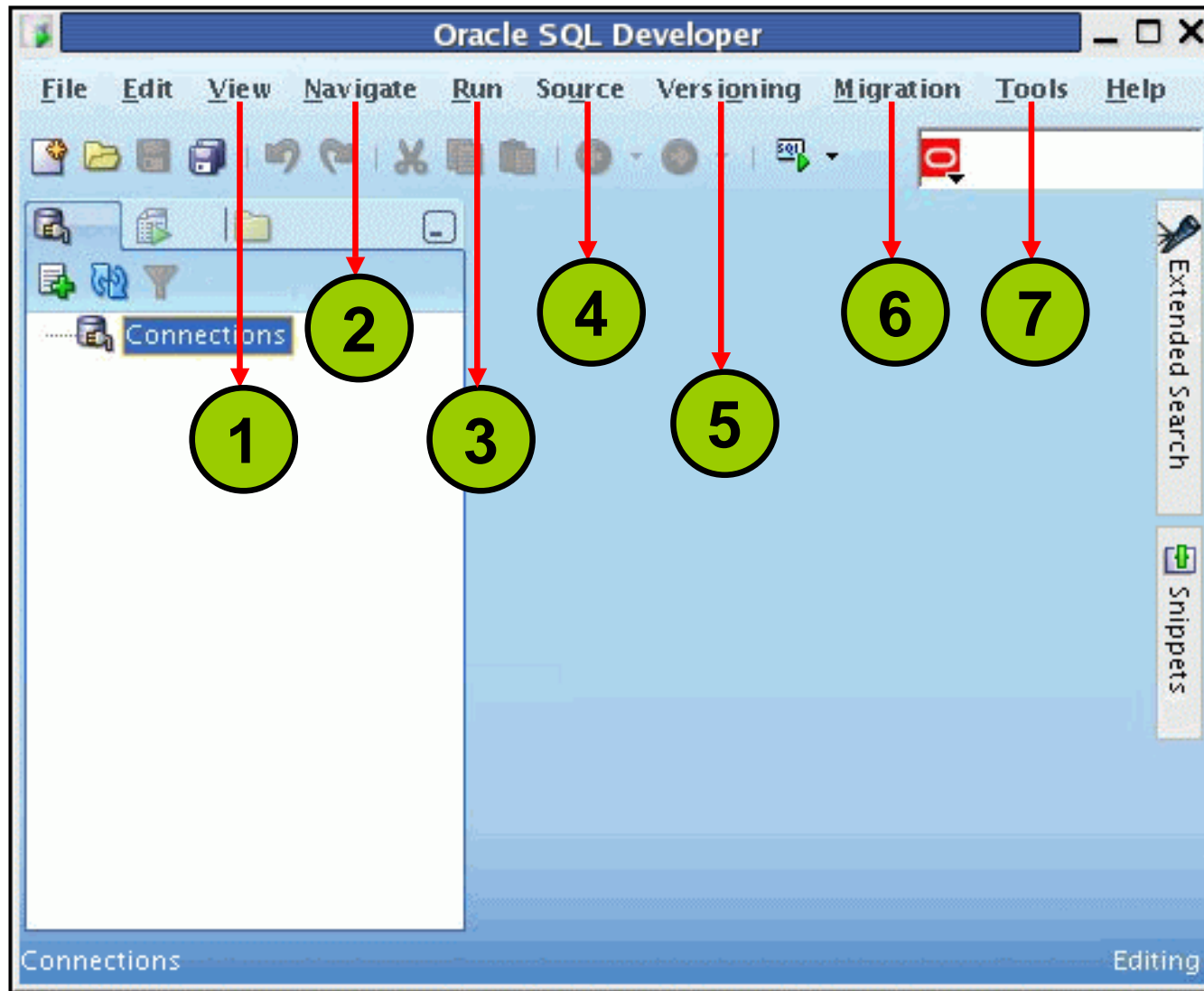
- Was developed in Java
- Supports Windows, Linux, and Mac OS X platforms
- Uses the JDBC Thin driver for default connectivity
- Does not require an installer
- Connects to any Oracle Database version 9.2.0.1 and later
- Is bundled with JRE 1.5

# Installing SQL Developer

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



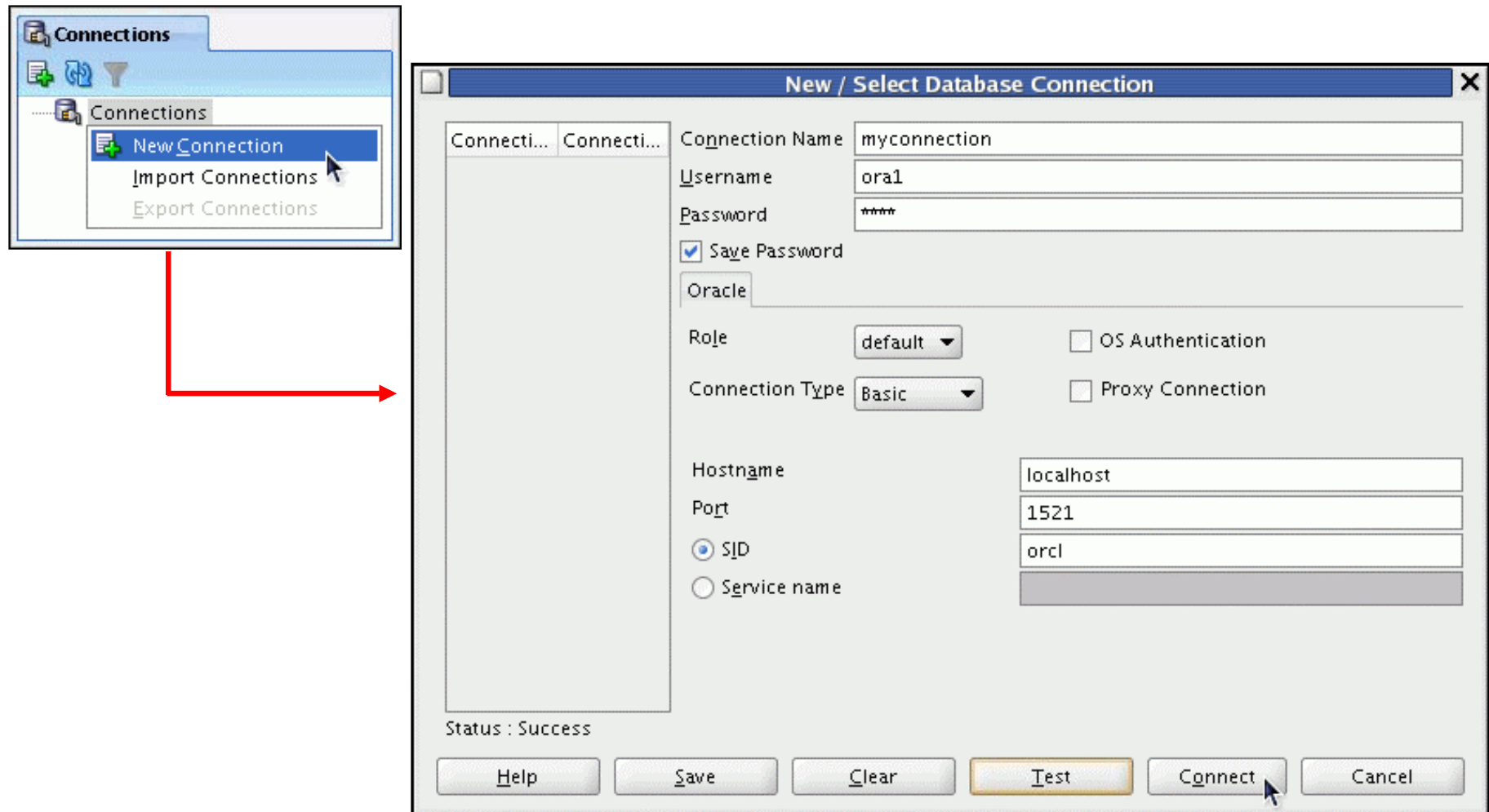
# Menus for SQL Developer



# 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.

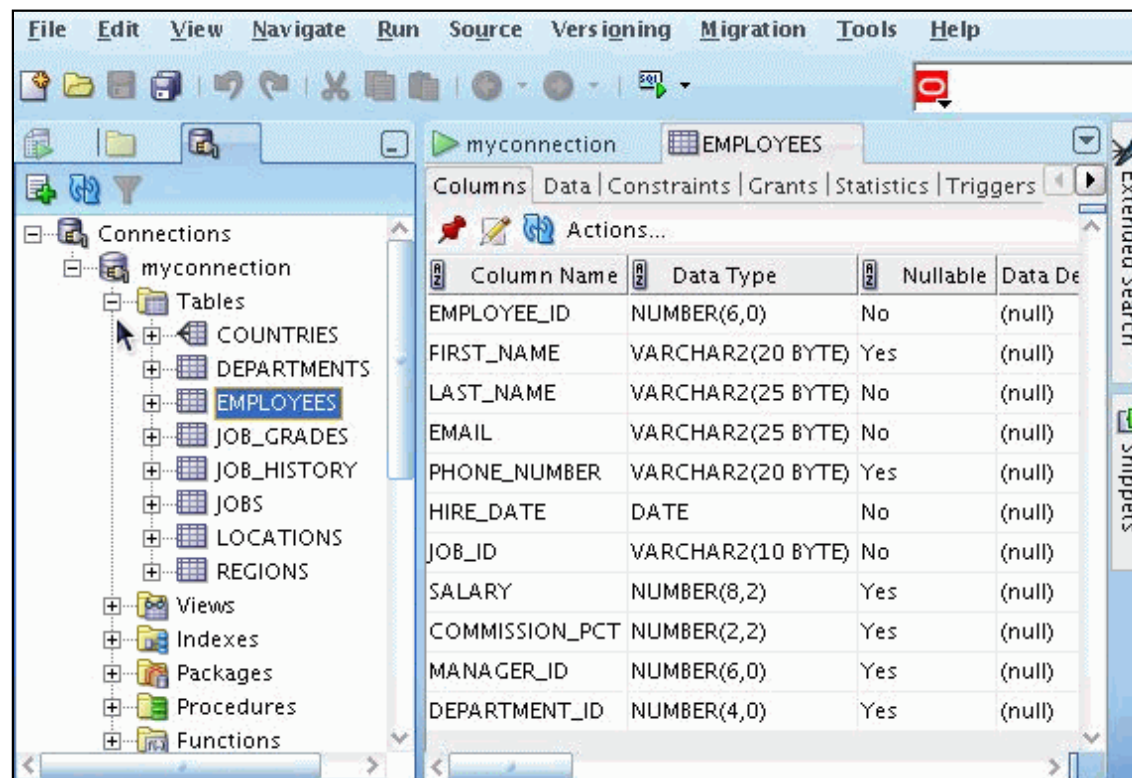
# Creating a Database Connection



# Browsing Database Objects

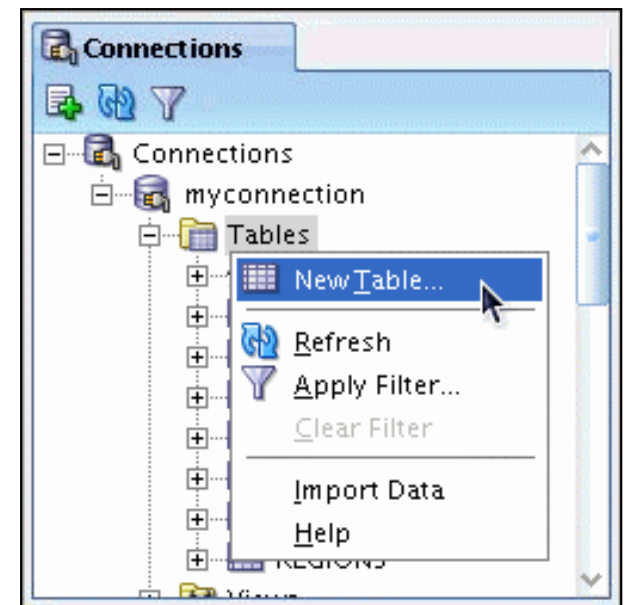
Use the Database Navigator to:

- Browse through many objects in a database schema
- Do a quick review of the definitions of objects



# Creating a Schema Object

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





# Creating a New Table: Example

**Create Table**

Schema:  ☒ Advanced

Name:

Table Type: ☒ Normal ☐ External ☐ Index Organized ☐ Temporary (Transaction) ☐ Temporary (Session)

**Columns:**

- DEPARTMENT\_ID
- DEPARTMENT\_NAME
- LOCATION\_ID
- MANAGER\_ID
- COLUMN1**

**Column Properties**

Name:

Datatype: ☒ Simple ☐ Complex

Type:

Size:

Units:

Default:

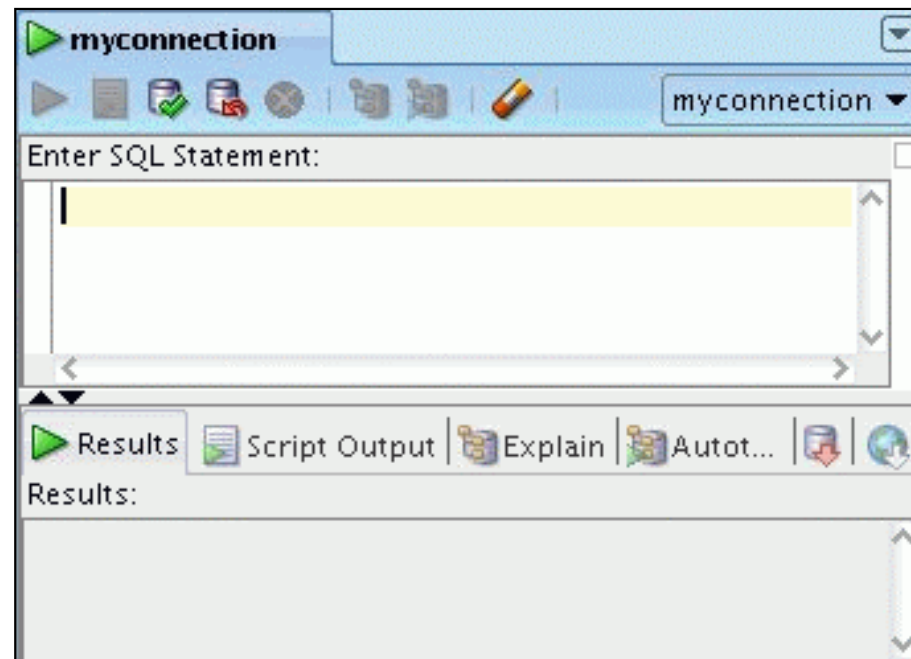
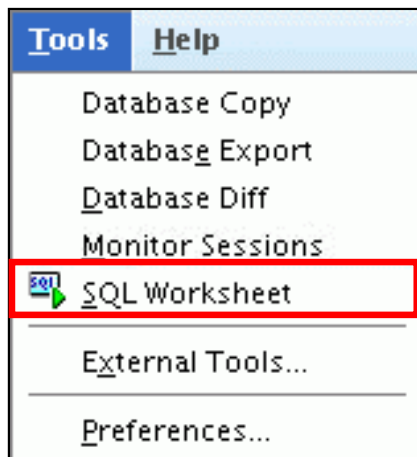
☐ Cannot be NULL

Comment:

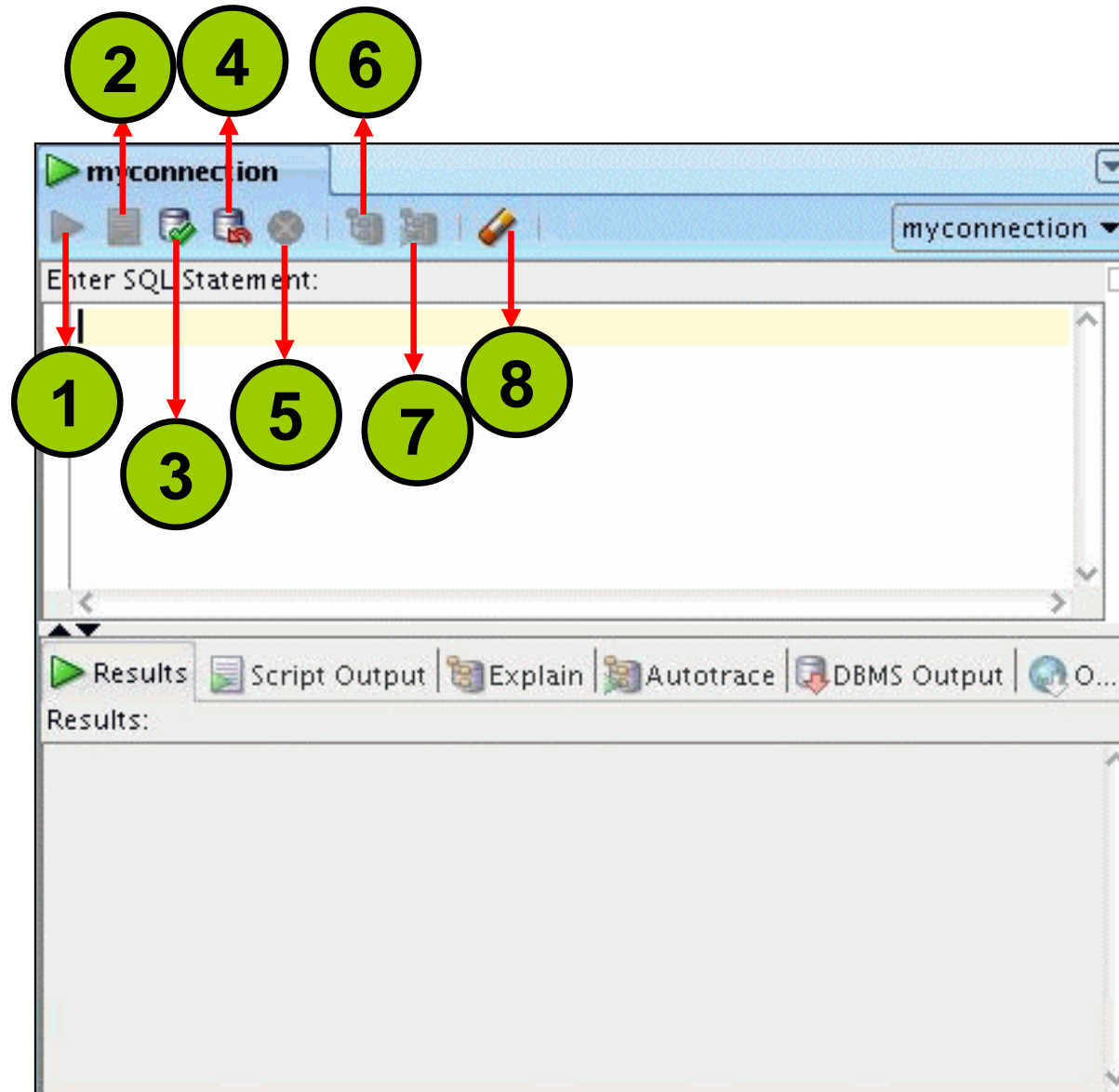
Help OK Cancel

# Using SQL Worksheet

- Use 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.

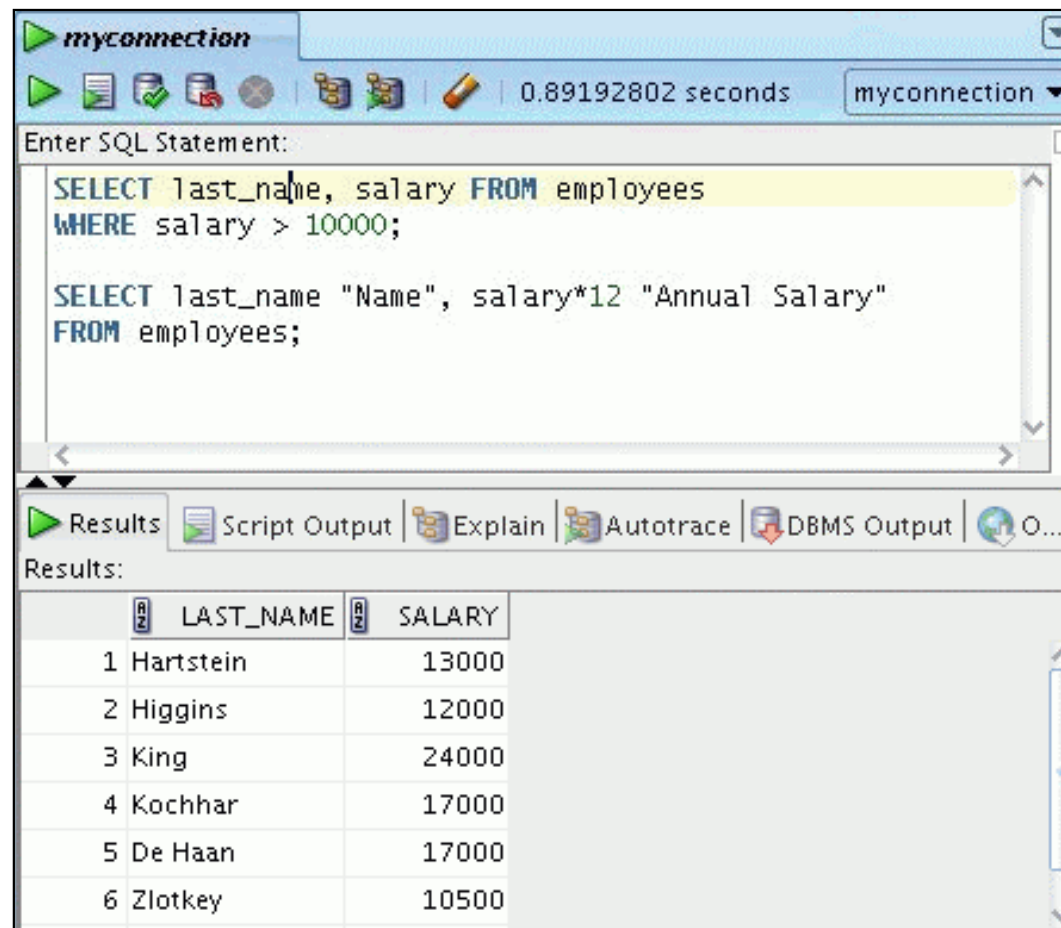


# Using SQL Worksheet

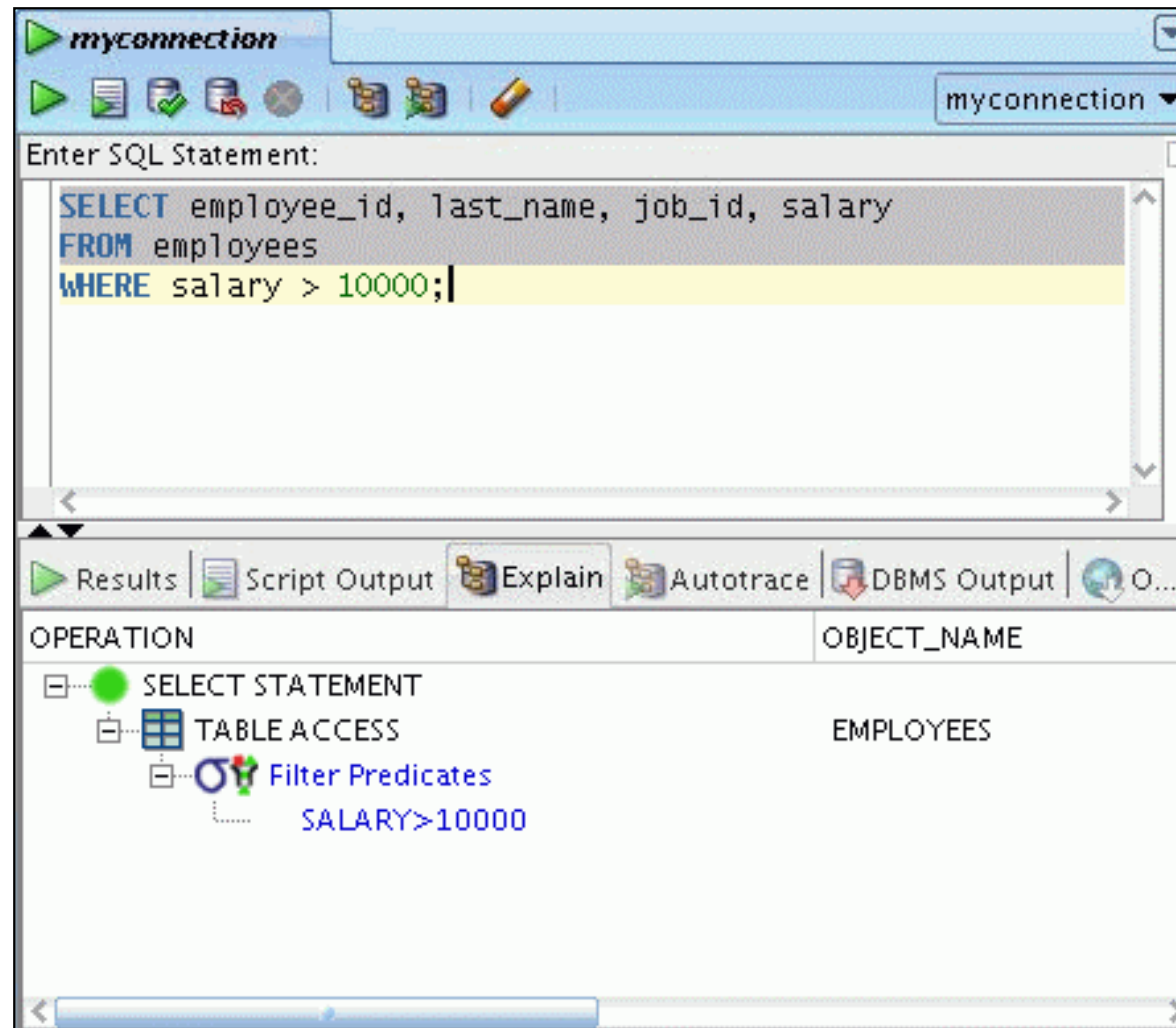


# Executing SQL Statements

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



# Viewing the Execution Plan



The screenshot shows the Oracle SQL Developer interface. The top pane, titled "myconnection", contains the SQL statement:

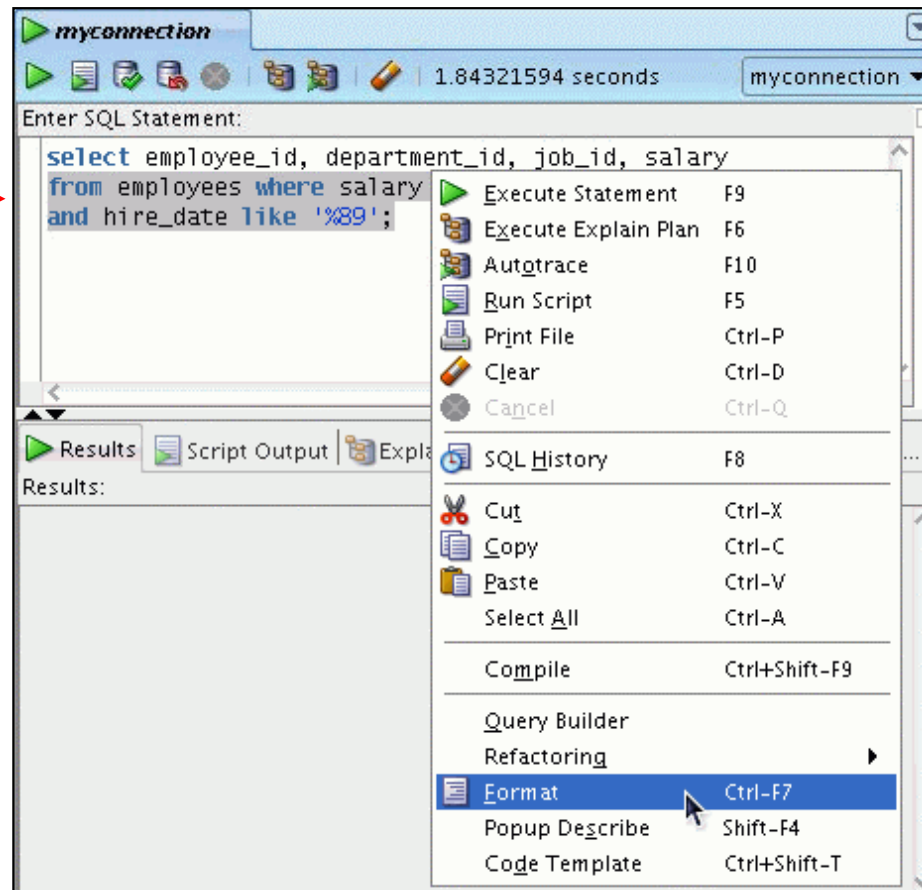
```
SELECT employee_id, last_name, job_id, salary
FROM employees
WHERE salary > 10000;
```

The bottom pane shows the execution plan. The tabs at the top of this pane are "Results", "Script Output", "Explain", "Autotrace", "DBMS Output", and "O...". The "Explain" tab is selected, displaying the following execution plan:

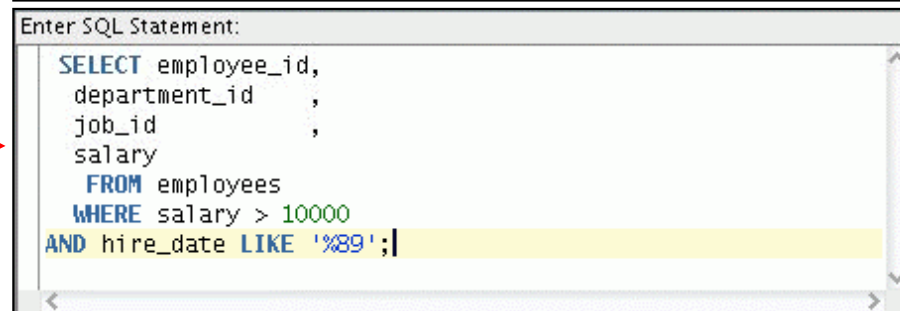
OPERATION	OBJECT_NAME
SELECT STATEMENT	
TABLE ACCESS	EMPLOYEES
Filter Predicates	
SALARY>10000	

# Formatting the SQL Code

**Before  
Formatting**

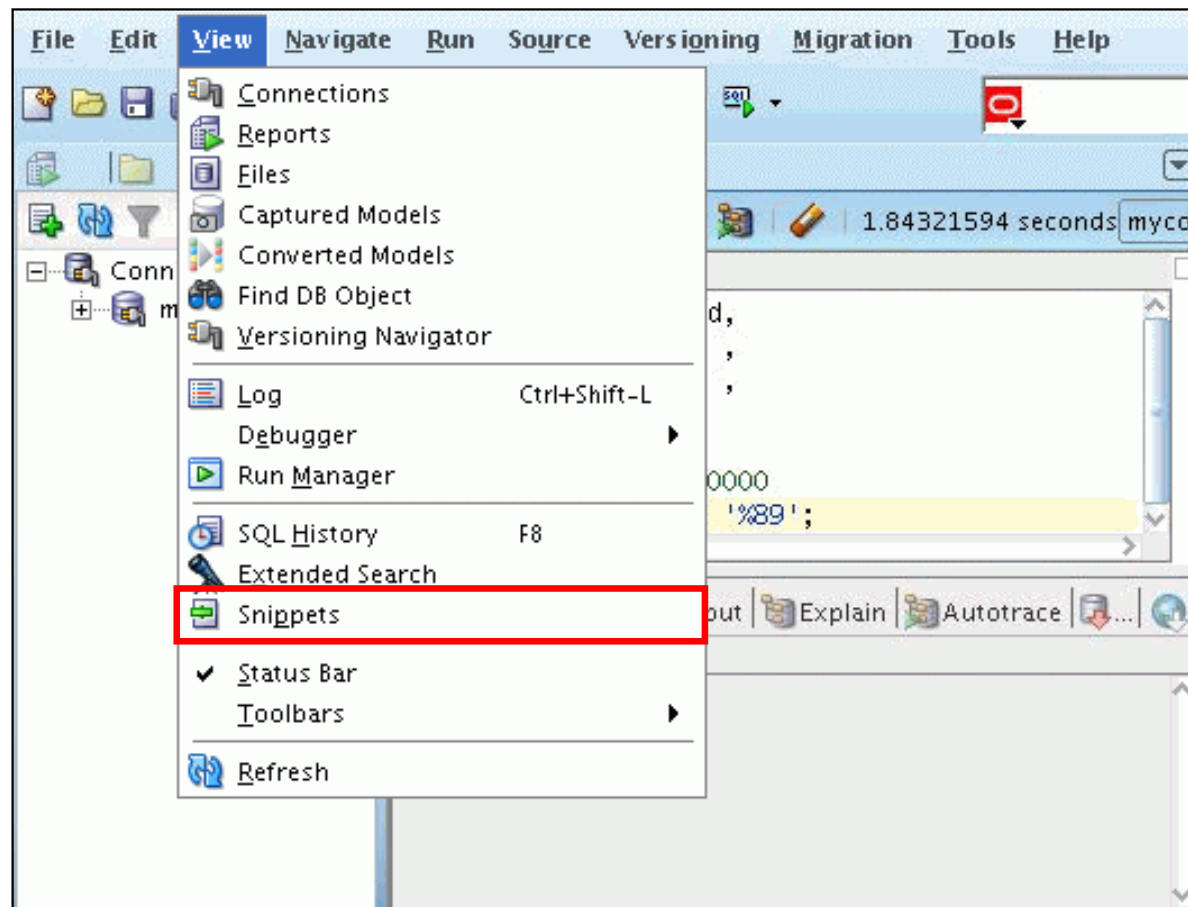


**After  
Formatting**



# Using Snippets

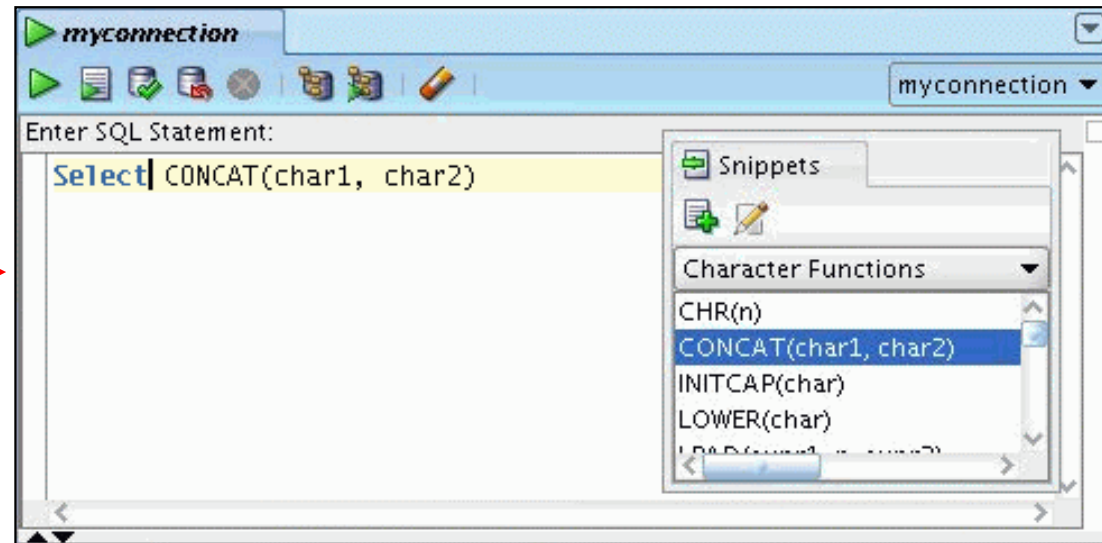
Snippets are code fragments that may be just syntax or examples.



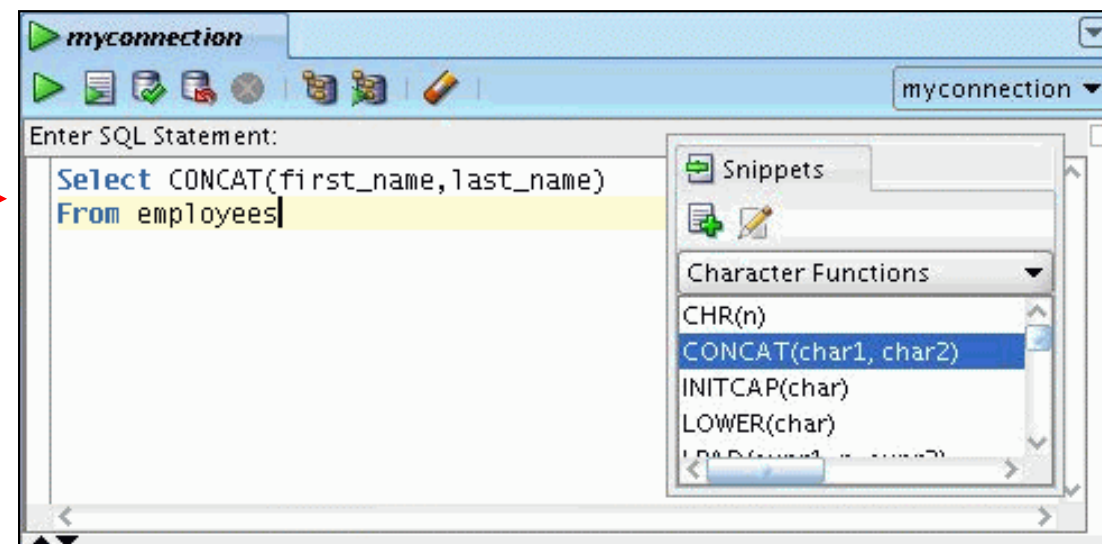


# Using Snippets: Example

Inserting a snippet



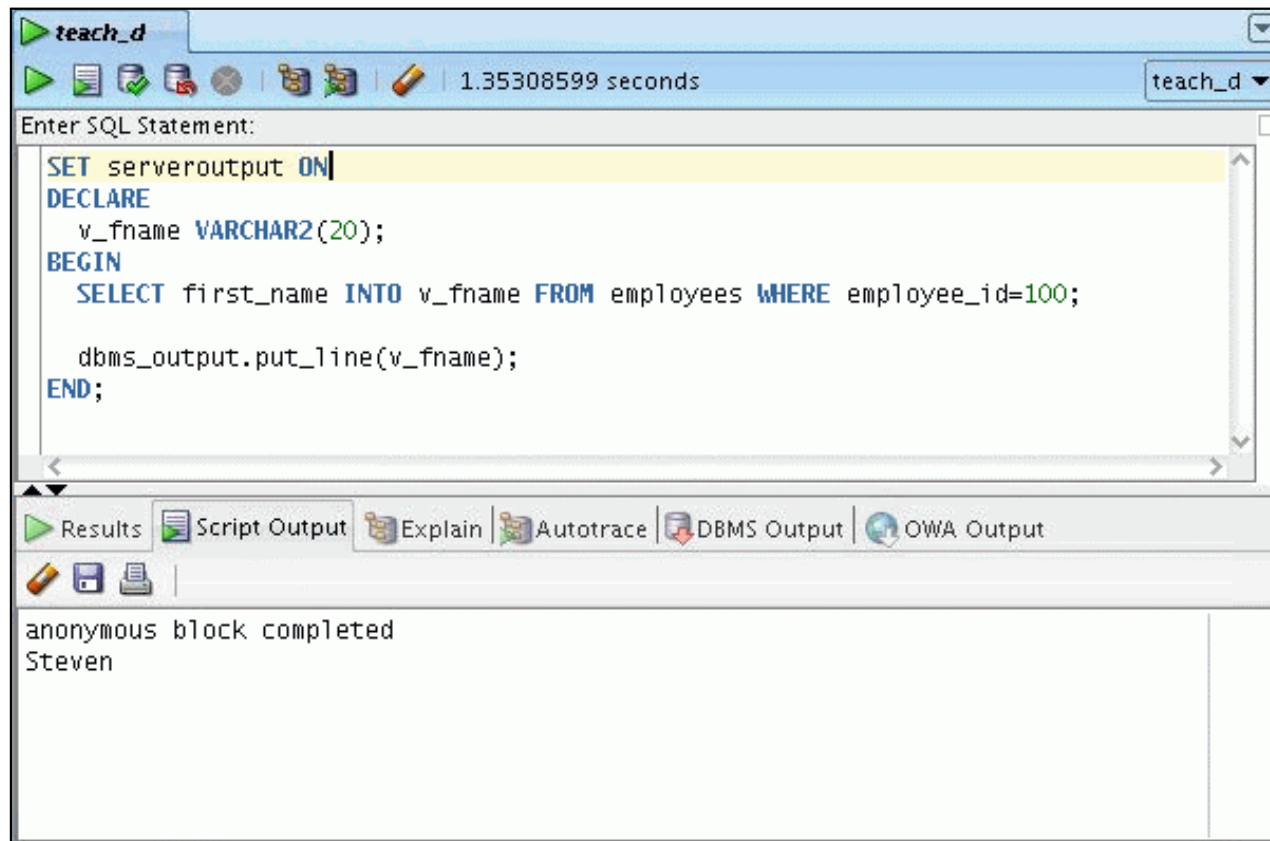
Editing the snippet





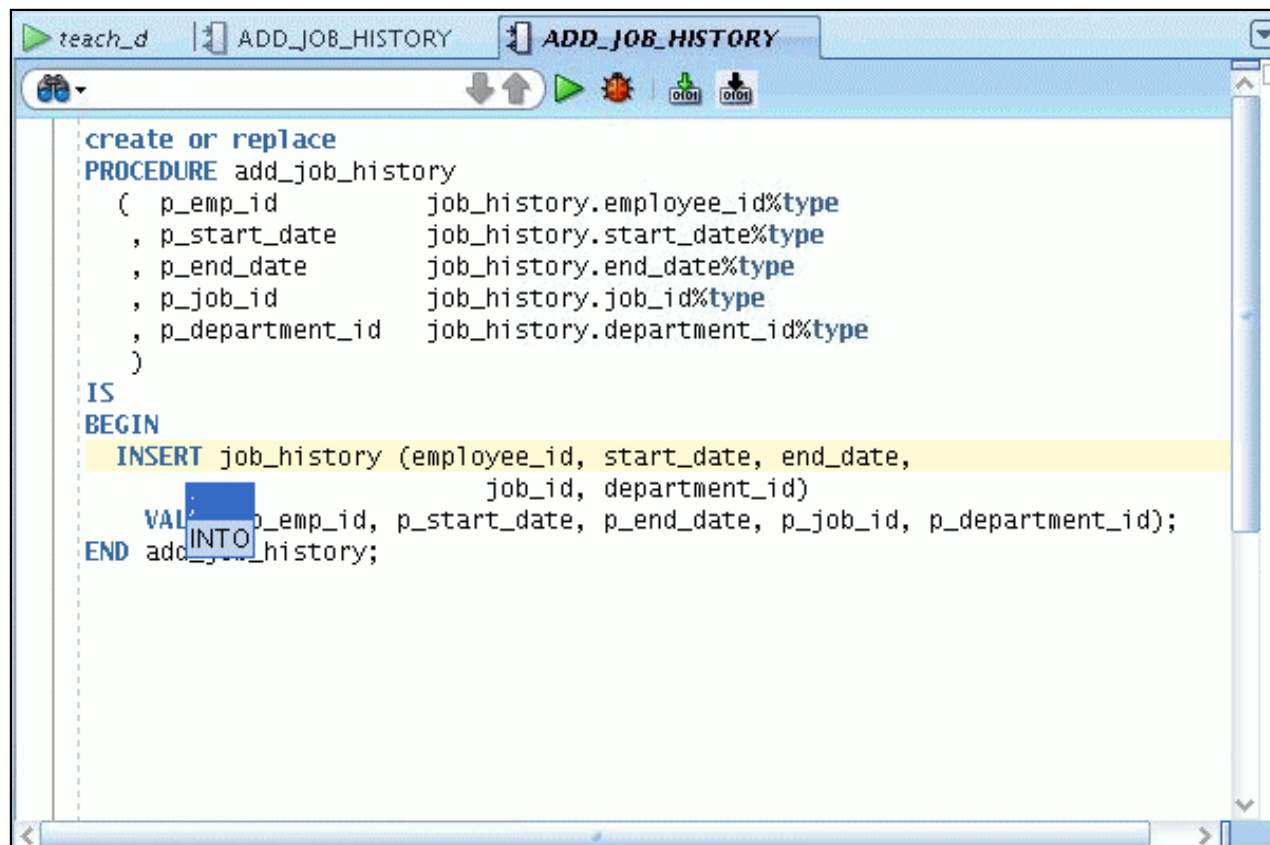
# Creating an Anonymous Block

Create an anonymous block and display the output of DBMS\_OUTPUT package statements.



# Editing the PL/SQL Code

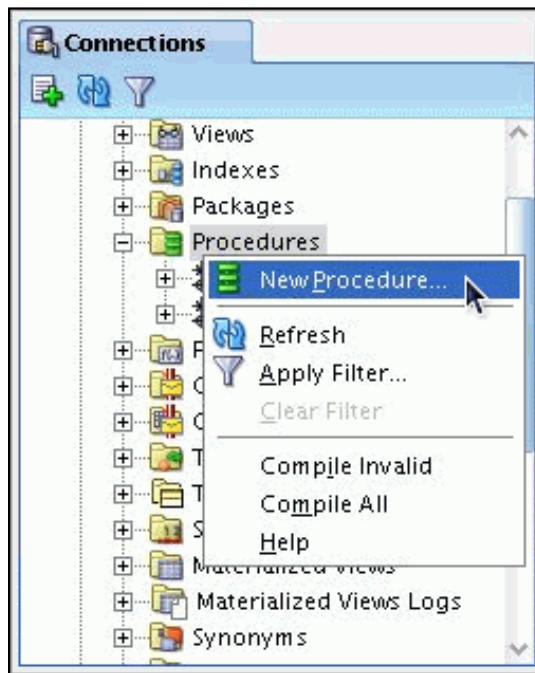
Use the full-featured editor for PL/SQL program units:



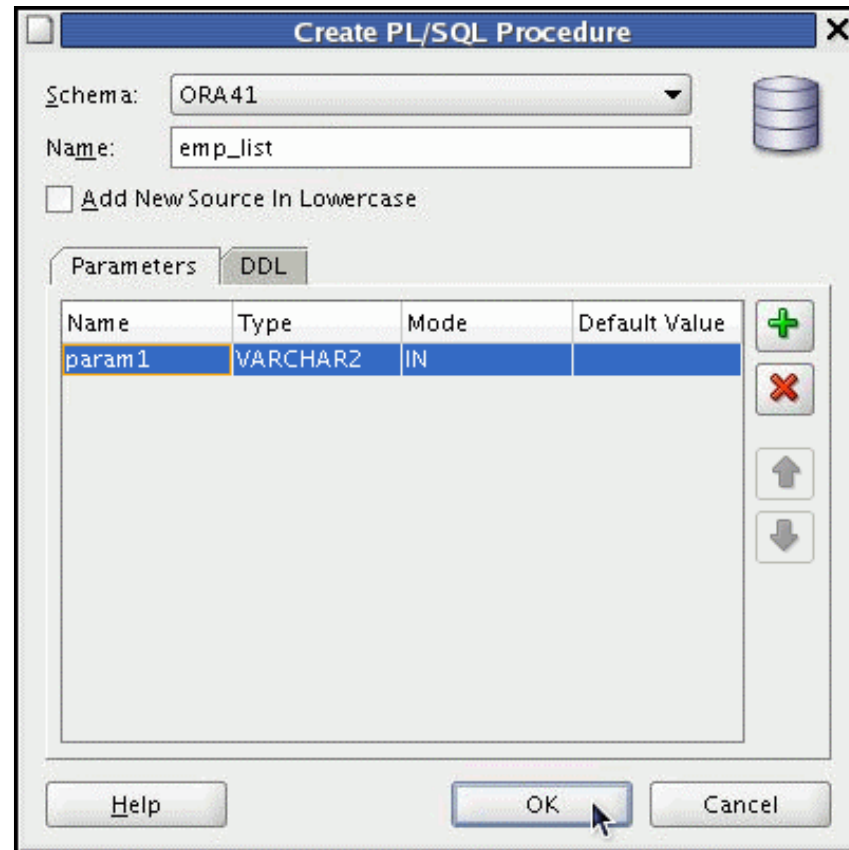
The screenshot shows the Oracle SQL Developer interface. The top toolbar includes icons for running, debugging, and saving. The main editor window displays the following PL/SQL code:

```
create or replace
PROCEDURE add_job_history
( p_emp_id          job_history.employee_id%type
, p_start_date      job_history.start_date%type
, p_end_date        job_history.end_date%type
, p_job_id          job_history.job_id%type
, p_department_id   job_history.department_id%type
)
IS
BEGIN
  INSERT job_history (employee_id, start_date, end_date,
                    job_id, department_id)
VALUES (p_emp_id, p_start_date, p_end_date, p_job_id, p_department_id);
END add_job_history;
```

# Creating a PL/SQL Procedure



1



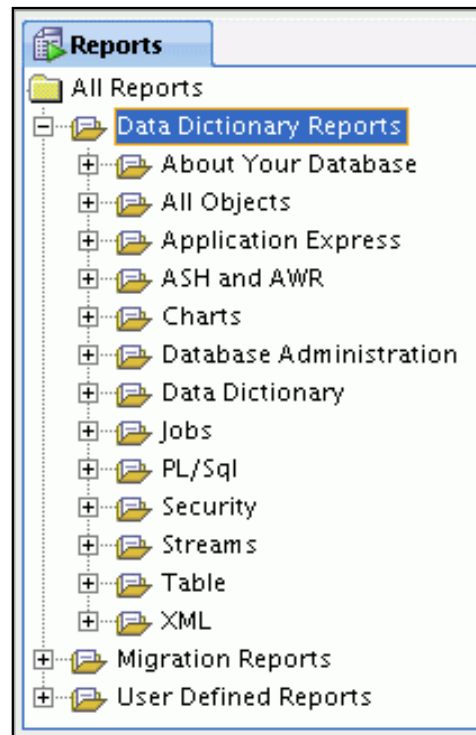
2

# Using SQL\*Plus

- SQL Worksheet does not support all SQL\*Plus statements.
- SQL\*Plus statements that are not supported by SQL Worksheet are:
  - append
  - archive
  - attribute
  - break
  - change
  - clear

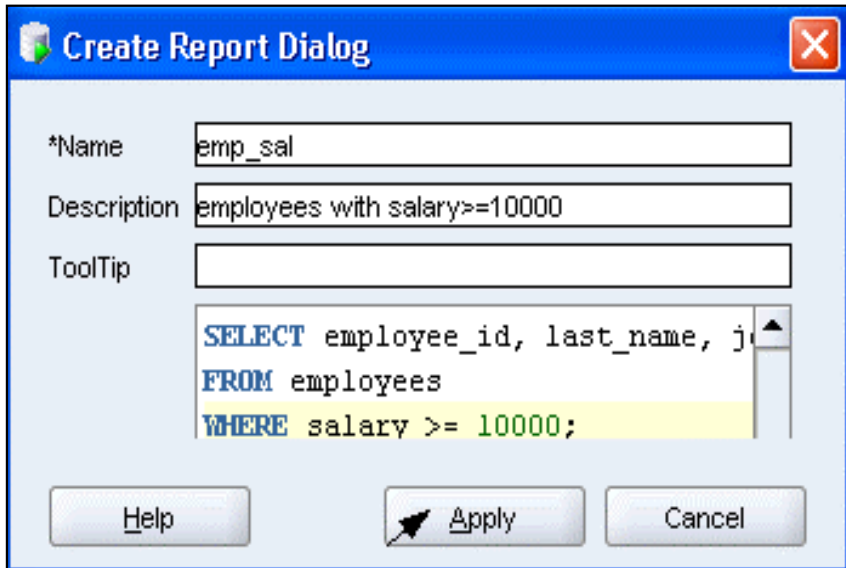
# Database Reporting

- SQL Developer provides you with a number of predefined reports about your database and objects.
- The Reports are organized into categories.
- You can create your own customized reports too.

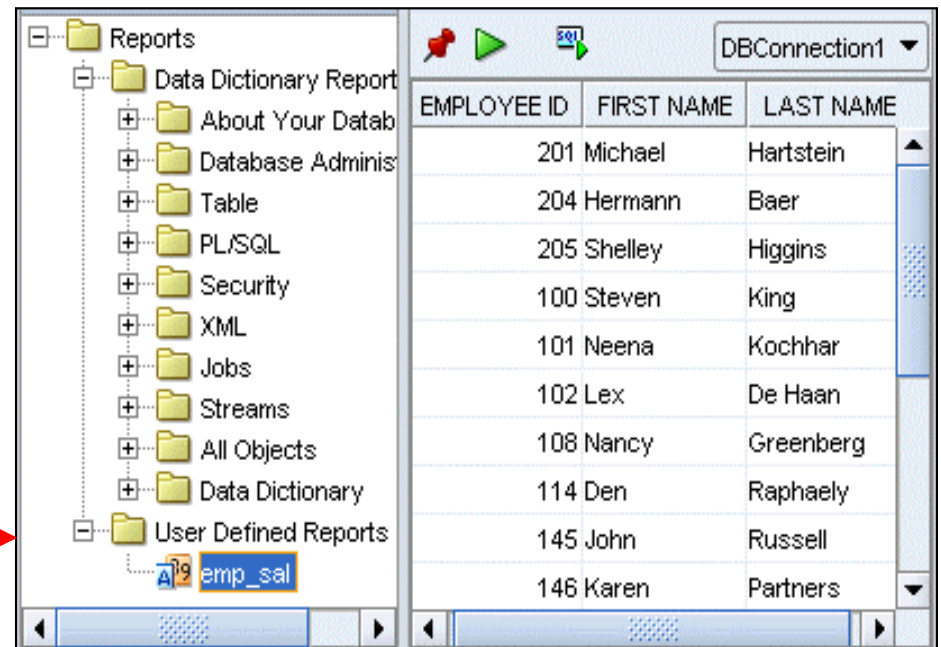


# Creating a User-Defined Report

Create and save user-defined reports for repeated use.



The 'Create Report Dialog' window is shown. It has a blue title bar with a close button. The fields are: \*Name: emp\_sal, Description: employees with salary >= 10000, ToolTip: (empty). The SQL query is: `SELECT employee_id, last_name, job_id  
FROM employees  
WHERE salary >= 10000;` The 'Apply' button is highlighted with a red arrow pointing to the 'User Defined Reports' folder in the right-hand window.



The Oracle Reports Desktop interface is shown. The left pane displays a tree view of reports, with 'User Defined Reports' expanded and 'emp\_sal' selected. The right pane shows a table of employee data.

EMPLOYEE ID	FIRST NAME	LAST NAME
201	Michael	Hartstein
204	Hermann	Baer
205	Shelley	Higgins
100	Steven	King
101	Neena	Kochhar
102	Lex	De Haan
108	Nancy	Greenberg
114	Den	Raphaely
145	John	Russell
146	Karen	Partners

# 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 SQL Worksheet
- Create and save custom reports