Oracle Database 10g: PL/SQL Fundamentals

Electronic Presentation
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:

• Appreciate that 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 that are to be covered on day 1:

1. Introduction
2. Introduction to PL/SQL
3. Declaring PL/SQL Variables
4. Creating the Executable Section
5. Interacting with the Oracle Database Server
6. Writing Control Structures
Lessons that are to be covered on day 2:
6. Working with Composite Data Types
7. Using Explicit Cursors
8. Including Exception Handling
9. Creating Stored Procedures and Functions
The Human Resources (hr) Data Set
Oracle10g
Oracle Database 10g

Object Relational Data

Documents

Multimedia

Messages
Oracle Application Server 10g

Portals

Transactional Apps

Business intelligence

Integration

Application development framework

Application server

Copyright © 2004, Oracle. All rights reserved.
Oracle Enterprise Manager 10g
Grid Control

- Software provisioning
- Application service-level monitoring
Oracle Internet Platform

Clients
Any browser
Any mail client
Any FTP client

Internet applications

Business logic and data

Presentation and business logic

Databases
Application servers

Network services

System management

Development tools

SQL
PL/SQL
Java
Summary

In this lesson, you should have learned how to:

• Describe the course objectives and course agenda
• Identify the 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
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 iSQL*Plus as a development environment for PL/SQL
• Output messages in 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
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

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

Database Server Constructs
- Anonymous blocks
- Stored procedures or functions
- Stored packages
- Database triggers
- Object types
PL/SQL Programming Environments
PL/SQL Programming Environments

*iSQL*Plus

![*SQL*Plus](image)
PL/SQL Programming Environments

Workspace

Enter SQL, PL/SQL and SQL*Plus statements.
iSQL*Plus Architecture
Create an Anonymous Block

Type the anonymous block in the iSQL*Plus workspace:

```sql
DECLARE
  f_name VARCHAR20;
BEGIN
  SELECT first_name INTO f_name FROM employees WHERE employee_id=100;
END;
```
Execute an Anonymous Block

Click the Execute button to execute the anonymous block:

```
DECLARE
  f_name VARCHAR20;

BEGIN
  SELECT first_name INTO f_name FROM employees WHERE employee_id=100;
END;
```

PL/SQL procedure successfully completed.
Test the Output of a PL/SQL Block

- Enable output in iSQL*Plus with the command
  `SET SERVEROUTPUT ON`
- Use a predefined Oracle package and its procedure:
  - `DBMS_OUTPUT.PUT_LINE`

```
SET SERVEROUTPUT ON
...
DBMS_OUTPUT.PUT_LINE(' The First Name of the Employee is ' || f_name);
...
```
Test the Output of a PL/SQL Block

```sql
SET SERVEROUTPUT ON

DECLARE
  f_name VARCHAR(20);

BEGIN
  SELECT first_name INTO f_name FROM employees WHERE employee_id=100;
  DBMS_OUTPUT.PUT_LINE('The First Name of the Employee is ' || f_name);
END;
```

The First Name of the Employee is Steven

PL/SQL procedure successfully completed.
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 iSQL*Plus 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 %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

```
SELECT first_name, department_id
INTO emp_fname, emp_deptno
FROM ...
```
Identifiers

Identifiers are used for:

- Naming a variable
- Providing a convention 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;
   /

Copyright © 2004, Oracle. All rights reserved.
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;
/

3rd Sunday in June is : Father's day
2nd Sunday in May is : Mother's day
PL/SQL procedure successfully completed.
Types of Variables

• PL/SQL variables:
  – Scalar
  – Composite
  – Reference
  – Large objects (LOB)

• Non-PL/SQL variables: Bind variables
Types of Variables

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

```
Mynace VARCHAR2(20) := 'John';
Mynace VARCHAR2(20) DEFAULT 'John';
```

• Declare one identifier per line for better readability and code maintenance.
Guidelines for Declaring PL/SQL Variables

• Avoid using column names as identifiers.

```sql
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 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 (Institute of Electrical and Electronics Engineers) 754 format
- Offer better interoperability and operational speed
- Store values beyond the values that the data type NUMBER can store
- Offer 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;
...
```
The %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 logical operators **AND**, **OR**, and 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:

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

```sql
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. Enter value for empno: 100

2. old 2: empno NUMBER(6):= &empno;
   new 2: empno NUMBER(6):= 100;
   PL/SQL procedure successfully completed.

   EMP_SALARY
   24000

3. PL/SQL procedure successfully completed.

   EMP_SALARY
   24000
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;
/

Input Required

Please enter a valid employee number: 100
Using `DEFINE` for 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

PL/SQL table structure

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SMITH</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>JONES</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>NANCY</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>TIM</td>
<td></td>
</tr>
</tbody>
</table>

PL/SQL table structure

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5000</td>
</tr>
<tr>
<td>2</td>
<td>2345</td>
</tr>
<tr>
<td>3</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>3456</td>
</tr>
</tbody>
</table>

PLS_INTEGER

VARCHAR2

NUMBER

Copyright © 2004, Oracle. All rights reserved.
LOB Data Type Variables

- Book (CLOB)
- Photo (BLOB)
- Movie (BFILE)
- NCLOB
Summary

In this lesson, you should have learned how to:

• Identify valid and invalid identifiers
• Declare variables in the declarative section of a PL/SQL block
• Initialize variables and utilize them in the executable section
• Differentiate between scalar and composite data types
• Use the %TYPE attribute
• Make use of 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
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, digits, 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.
    
    ```plsql
    name := 'Henderson';
    ```
  - Numbers can be simple values or scientific notation.

- **Statements can continue over several lines.**
Commenting Code

• Prefix single-line comments with two dashes (--).
• Place multiple-line comments between the symbols “/∗” and “∗/”.

Example:

```sql
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:
  ```sql
  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:
  ```sql
  emp_name:= LOWER(emp_name);
  ```
Data Type Conversion

- 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:

```sql
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;/

1

2

Copyright © 2004, Oracle. All rights reserved.
Qualify an Identifier

<<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);
    DBMS_OUTPUT.PUT_LINE('Date of Birth: '||date_of_birth);
  END;
END;
/
Determining Variable Scope

<<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;
/

1
2
Operators in PL/SQL

- Logical
- Arithmetic
- Concatenation
- Parentheses to control order of operations

Same as in SQL

- Exponential operator (**)

Copyright © 2004, Oracle. All rights reserved.
Operators in PL/SQL

Examples:

- Increment the counter for a loop.
  
  ```plsql
  loop_count := loop_count + 1;
  ```

- Set the value of a Boolean flag.
  
  ```plsql
  good_sal := sal BETWEEN 50000 AND 150000;
  ```

- Validate whether an employee number contains a value.
  
  ```plsql
  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:

```sql
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 you should 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
Interacting with the Oracle Server
Objectives

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

- Decide 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
- Make use of 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:

```sql
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;
/
```
Retrieve the **hire_date** and the **salary** for the specified employee.

**Example:**

```sql
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:

```sql
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;
/
```

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

```sql
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:

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

Insert or update rows in the \texttt{copy_emp} table to match the \texttt{employees} table.

```sql
DECLARE
  empno employees.employee_id\%TYPE := 100;
BEGIN
  MERGE INTO copy_emp c
  USING employees e
  ON (e.employee_id = 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 cursors: Created and managed internally by the Oracle server to process SQL statements
  – Explicit cursors: Explicitly declared by the programmer
**SQL Cursor Attributes for Implicit Cursors**

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

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL%FOUND</td>
<td>Boolean attribute that evaluates to <code>TRUE</code> if the most recent SQL statement returned at least one row.</td>
</tr>
<tr>
<td>SQL%NOTFOUND</td>
<td>Boolean attribute that evaluates to <code>TRUE</code> if the most recent SQL statement did not return even one row.</td>
</tr>
<tr>
<td>SQL%ROWCOUNT</td>
<td>An integer value that represents number of rows affected by the most recent SQL statement.</td>
</tr>
</tbody>
</table>
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:

```sql
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
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
- Make use of guidelines while using the conditional control structures
Controlling Flow of Execution

IF... THEN... END IF;

IF... THEN... ELSE... END IF;

IF... THEN... ELSIF... THEN... END IF;

IF... THEN... ELSIF... THEN... ELSE... END IF;

CASE WHEN... THEN... WHEN... THEN... WHEN... THEN... ELSE END CASE;

loop

for

while
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 IF;
END;
/

PL/SQL procedure successfully 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;
/

I am not a child
PL/SQL procedure successfully completed.
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;
/

I am in my thirties
PL/SQL procedure successfully completed.
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 IF;
END;
/

I am not a child
PL/SQL procedure successfully completed.
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.

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

```sql
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.
• In conditional control statements, if the condition yields NULL, its associated sequence of statements is not executed.
Logic Tables

Build a simple Boolean condition with a comparison operator.

<table>
<thead>
<tr>
<th>AND</th>
<th>TRUE</th>
<th>FALSE</th>
<th>NULL</th>
<th>OR</th>
<th>TRUE</th>
<th>FALSE</th>
<th>NULL</th>
<th>NOT</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>TRUE</td>
<td>FALSE</td>
<td>NULL</td>
<td>TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
</tr>
<tr>
<td>FALSE</td>
<td>FALSE</td>
<td>FALSE</td>
<td>FALSE</td>
<td>FALSE</td>
<td>TRUE</td>
<td>FALSE</td>
<td>NULL</td>
<td>FALSE</td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
<td>FALSE</td>
<td>NULL</td>
<td>NULL</td>
<td>TRUE</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
</tr>
</tbody>
</table>
Boolean Conditions

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

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

<table>
<thead>
<tr>
<th>REORDER_FLAG</th>
<th>AVAILABLE_FLAG</th>
<th>FLAG</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>TRUE</td>
<td>?</td>
</tr>
<tr>
<td>TRUE</td>
<td>FALSE</td>
<td>?</td>
</tr>
<tr>
<td>NULL</td>
<td>TRUE</td>
<td>?</td>
</tr>
<tr>
<td>NULL</td>
<td>FALSE</td>
<td>?</td>
</tr>
</tbody>
</table>
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:

```sql
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 While Using Loops

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

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

... BEGIN
  <<Outer_loop>>
  LOOP
    counter := counter+1;
    EXIT WHEN counter>10;
  END LOOP Outer_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;
/

Copyright © 2004, Oracle. All rights reserved.
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 using the **IF** statement
• Performing iterative steps 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 difference between 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 same data type.
PL/SQL Records

- Must contain one or more components of any scalar, RECORD, or INDEX BY table data type, called fields
- 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. \[
\text{\textbf{TYPE} type\_name IS RECORD}\]
   \[
   (\text{field\_declaration[, field\_declaration]}\ldots);\]

2. \[
\text{identifier type\_name;}\]

field\_declaration:

\[
\text{field\_name \{}\]
   \[
   \text{field\_type | variable\%TYPE}
   \mid \text{table\_column\%TYPE | table\%ROWTYPE}\}
\[
[[\text{NOT NULL}] \{}\]
   \[
   \text{:= | DEFAULT}\} \text{ expr}\]
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:

<table>
<thead>
<tr>
<th>Field1 (data type)</th>
<th>Field2 (data type)</th>
<th>Field3 (data type)</th>
</tr>
</thead>
<tbody>
<tr>
<td>employee_id</td>
<td>last_name</td>
<td>job_id</td>
</tr>
<tr>
<td>number(6)</td>
<td>varchar2(25)</td>
<td>varchar2(10)</td>
</tr>
<tr>
<td>100</td>
<td>King</td>
<td>AD_PRES</td>
</tr>
</tbody>
</table>
The `%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.
• The number and data types of the underlying database column may change at run time.
• The attribute is useful when retrieving a row with the SELECT * statement.
DEFINE employee_number = 124

DECLARE
  emp_rec employees%ROWTYPE;
BEGIN
  SELECT * INTO emp_rec FROM employees WHERE employee_id = &employee_number;
  INSERT INTO retired_emps(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;
/

The %ROWTYPE Attribute
Inserting a Record 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 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 type integer or string
  – Column of scalar or record data type
• Are unconstrained in size. However the size depends on the values the key data type can hold.
Creating an **INDEX BY** Table

Syntax:

```sql
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.

```sql
... TYPE ename_table_type IS TABLE OF
    employees.last_name%TYPE
    INDEX BY PLS_INTEGER;
...
ename_table ename_table_type;
```
## INDEX BY Table Structure

<table>
<thead>
<tr>
<th>Unique Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1</td>
<td>Jones</td>
</tr>
<tr>
<td>5</td>
<td>Smith</td>
</tr>
<tr>
<td>3</td>
<td>Maduro</td>
</tr>
<tr>
<td>PLS_INTEGER</td>
<td>Scalar</td>
</tr>
</tbody>
</table>
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:

```sql
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
```
Example of `INDEX BY` Table of Records

```sql
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

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Bombay</td>
</tr>
<tr>
<td>2</td>
<td>Sydney</td>
</tr>
<tr>
<td>3</td>
<td>Oxford</td>
</tr>
<tr>
<td>4</td>
<td>London</td>
</tr>
<tr>
<td></td>
<td>.....</td>
</tr>
</tbody>
</table>

2 GB
<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Bombay</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>Sydney</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>Oxford</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>London</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>Tokyo</td>
</tr>
</tbody>
</table>
Summary

In this lesson, you should have learned how to:

• Define and reference PL/SQL variables of composite data types:
  – PL/SQL records
  – INDEX BY tables
  – INDEX BY table of records

• Define a PL/SQL record by using the %ROWTYPE attribute
Practice 6: Overview

This practice covers the following topics:

• Declaring \texttt{INDEX \texttt{BY} tables}
• Processing data by using \texttt{INDEX \texttt{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 an implicit and an explicit cursor
• Discuss when and why to use an explicit cursor
• Declare and control explicit cursors
• Use simple loop and cursor FOR loop to fetch data
• Declare and use cursors with parameters
• Lock rows using the FOR UPDATE clause
• Reference the current row with the WHERE CURRENT clause
About Cursors

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

- 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

Table

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Department</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>King</td>
<td>AD_PRES</td>
</tr>
<tr>
<td>101</td>
<td>Kochhar</td>
<td>AD_VP</td>
</tr>
<tr>
<td>102</td>
<td>De Haan</td>
<td>AD_VP</td>
</tr>
<tr>
<td>139</td>
<td>Seo</td>
<td>ST_CLERK</td>
</tr>
<tr>
<td>140</td>
<td>Patel</td>
<td>ST_CLERK</td>
</tr>
</tbody>
</table>
Controlling Explicit Cursors

- Declare
- Open
- Fetch
- Empty?
- Yes: Close
- No: Test for existing rows
  - Yes: Return to Fetch if rows are found
  - No: Load the current row into variables
  - Identify the active set
  - Create a named SQL area
  - Release the active set
Controlling Explicit Cursors

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

Syntax:

```sql
CURSOR cursor_name IS
    select_statement;
```

Examples:

```sql
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;
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;
    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 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;/

Copyright © 2004, Oracle. All rights reserved.
Cursors and Records

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

```sql
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.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%ISOPEN</td>
<td>Boolean</td>
<td>Evaluates to \texttt{TRUE} if the cursor is open</td>
</tr>
<tr>
<td>%NOTFOUND</td>
<td>Boolean</td>
<td>Evaluates to \texttt{TRUE} if the most recent fetch does not return a row</td>
</tr>
<tr>
<td>%FOUND</td>
<td>Boolean</td>
<td>Evaluates to \texttt{TRUE} if the most recent fetch returns a row; complement of %NOTFOUND</td>
</tr>
<tr>
<td>%ROWCOUNT</td>
<td>Number</td>
<td>Evaluates to the total number of rows returned so far</td>
</tr>
</tbody>
</table>
The %ISOPEN Attribute

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

Example:

```
IF NOT emp_cursor%ISOPEN THEN
   OPEN emp_cursor;
END IF;
LOOP
   FETCH emp_cursor...
```
Example of `%ROWCOUNT` and `%NOTFOUND`

```sql
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

No need to declare the cursor.

Example:

```sql
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 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);
    ...

The 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.
The **WHERE CURRENT OF** Clause

Syntax:

```sql
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.

```sql
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;
...```

In this lesson, you should have learned how to:

- **Distinguish cursor types:**
  - Implicit cursors: Used for all DML statements and single-row queries
  - Explicit cursors: Used for queries of zero, one, or more rows
- **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 cursor with parameters
• Using the **FOR UPDATE** and **WHERE CURRENT OF** clauses
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

```
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

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;
/

Copyright © 2004, Oracle. All rights reserved.
Handling Exceptions with PL/SQL

- An exception is an error PL/SQL 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 raised
- Is the exception trapped?
  - yes: Execute statements in the `EXCEPTION` section
  - no: Terminate abruptly
- Propagate the exception
- Terminate gracefully
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

- **Declare**
- **Associate**
- **Reference**

**Declarative section**
- Name the exception
- Code `PRAGMA EXCEPTION_INIT`

**EXCEPTION section**
- Handle the raised exception
Non-Predefined Error

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;
/

1. insert_excep
2. PRAGMA EXCEPTION_INIT
3. WHEN insert_excep
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:

```sql
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

- **Declare (Declarative section)**: Name the exception.
- **Raise (Executable section)**: Explicitly raise the exception by using the `RAISE` statement.
- **Reference (Exception-handling section)**: Handle the raised exception.
Trapping User-Defined Exceptions

...  
ACCEPT deptno PROMPT 'Please enter the department number:'  
ACCEPT name   PROMPT 'Please enter the department name:'
DECLARE
   invalid_department EXCEPTION;
   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;
   END IF;
   COMMIT;
EXCEPTION
   WHEN invalid_department THEN
      DBMS_OUTPUT.PUT_LINE('No such department id.');
   END;
/

1. invalid_department EXCEPTION;
2. RAISE invalid_department;
3. WHEN invalid_department THEN
## Calling Environments

<table>
<thead>
<tr>
<th>Calling Environment</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>i</em>SQL*Plus</td>
<td>Displays error number and message to screen</td>
</tr>
<tr>
<td>Procedure Builder</td>
<td>Displays error number and message to screen</td>
</tr>
<tr>
<td>Oracle Developer Forms</td>
<td>Accesses error number and message in an ON-ERROR trigger by means of the ERROR_CODE and ERROR_TEXT packaged functions</td>
</tr>
<tr>
<td>Precompiler application</td>
<td>Accesses exception number through the SQLCA data structure</td>
</tr>
<tr>
<td>An enclosing PL/SQL block</td>
<td>Traps exception in exception-handling routine of enclosing block</td>
</tr>
</tbody>
</table>
Propagating Exceptions in a Subblock

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

```sql
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;
/```
The `RAISE_APPLICATION_ERROR` Procedure

Syntax:

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

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

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

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

Exception section:
   ...
   EXCEPTION
      WHEN NO_DATA_FOUND THEN
         RAISE_APPLICATION_ERROR (-20201,
            'Manager is not a valid employee.');
   END;
Summary

In this lesson, you should have learned 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

<table>
<thead>
<tr>
<th>Anonymous Blocks</th>
<th>Subprograms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unnamed PL/SQL blocks</td>
<td>Named PL/SQL blocks</td>
</tr>
<tr>
<td>Compiled every time</td>
<td>Compiled only once</td>
</tr>
<tr>
<td>Not stored in the database</td>
<td>Stored in the database</td>
</tr>
<tr>
<td>Cannot be invoked by other apps</td>
<td>They are named and therefore can be invoked by other apps</td>
</tr>
<tr>
<td>Do not return values</td>
<td>Subprograms called functions must return values</td>
</tr>
<tr>
<td>Cannot take parameters</td>
<td>Can take parameters</td>
</tr>
</tbody>
</table>
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;

Inserted 1 row
PL/SQL procedure successfully completed.

<table>
<thead>
<tr>
<th>DEPARTMENT_ID</th>
<th>DEPARTMENT_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>280</td>
<td>ST-Curriculum</td>
</tr>
</tbody>
</table>
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;
/

Salary > average
PL/SQL procedure successfully completed.
Passing 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;
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
REF Cursors
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;
depth_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.
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;/

An Example of Fetching
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.
Connection Navigator
Application Navigator
Structure Window

```
mpackage1
Imports
MyJMSWebServiceStub
  MyJMSWebServiceStub()
  getEndPoint() : String
  main(String[] : void
  receive() : Vector
  send(String[] : void
  setEndPoint(String) : void
  _endpoint : String
  m_httpConnection : OracleSOAPHTTPConnection
  m_smr : SOAPMappingRegistry
```
Editor Window

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

```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];
    }
}
```

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

Skeleton of the function

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

Compilation with errors

Compilation without errors
Running a Program Unit

```
DECLARE
  X NUMBER;
  Y NUMBER;
BEGIN
  X := NULL;
  Y := NULL;
  OE.SWAP(
    X => X,
    Y => Y
  );
  DBMS_OUTPUT.PUT_LINE('X = ' || X);
  DBMS_OUTPUT.PUT_LINE('Y = ' || Y);
END;
```
Dropping a Program Unit

Drop Confirmation

Are you sure you want to drop PROCEDURE OE_TESTING?

Yes  No
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

On Windows, enter the path to the SQL*Plus executable. On UNIX, you also need to specify the xterm command. For example: /usr/bin/xterm -e /oracle/bin/sqlplus

Registered JDBC Drivers:

Driver Class:
Library: JClient Runtime

Copyright © 2004, Oracle. All rights reserved.
Setting Breakpoints

```sql
PROCEDURE "TEST_DEBUG" (p_cust_id IN NUMBER)
AS
v_cust customers%ROWTYPE;
BEGIN
  SELECT * INTO v_cust
  FROM customers
  WHERE customer_id = p_cust_id;
  dbms_output.put_line('Customer ID is ' || v_cust.customer_id);
  dbms_output.put_line('Customer Name is ' || v_cust.cust_first_name);
END;
```
Stepping Through Code

Debug Resume

PROCEDURE "TEST_DEBUG" (p_cust_id IN NUMBER) AS
  v_cust customers%ROWTYPE;
BEGIN
  SELECT * into v_cust
  FROM customers
  where customer_id = p_cust_id;
  dbms_output.put_line('Customer ID is ' || v_cust.customer_id);
  dbms_output.put_line('Customer Name is ' || v_cust.cust_first_name);
END;
Examining and Modifying Variables

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>P_CUST_ID</td>
<td>103</td>
<td>NUMBER</td>
</tr>
<tr>
<td>V_CUST</td>
<td></td>
<td>Rowtype</td>
</tr>
</tbody>
</table>

Data window
Examining and Modifying Variables

Smart Data window
Examining and Modifying Variables

Watches window
Examining and Modifying Variables

Stack window

<table>
<thead>
<tr>
<th>Class</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEST_DEBUG</td>
<td>TEST_DEBUG</td>
</tr>
<tr>
<td>ANONYMOUS</td>
<td>BLOCK</td>
</tr>
</tbody>
</table>
Examining and Modifying Variables

Classes window