Oracle Database 10*g*: Advanced PL/SQL

Student Guide

D17220GC10

Edition 1.0

June 2004

D39598

ORACLE®

Authors

Nancy Greenberg
Aniket Raut

**Technical Contributors
and Reviewers**

Andrew Brannigan
Christoph Burandt
Dairy Chan
Yanti Chang
Laszlo Czinkoczki
Janis Fleishman
Mark Fleming
Stefan Grenstad
Craig Hollister
Bryn Llewellyn
Yi L. Lu
Marcelo Manzano
Nagavalli Pataballa
Helen Robertson
John Soltani
S Matt Taylor Jr
Ric Van Dyke

Publisher

Poornima G

Contents

Preface

Appendix A

Appendix B

Appendix C

Appendix D

Preface

Preface - 2

Preface

Before You Begin This Course

Before you begin this course, you should have thorough knowledge of SQL, SQL*Plus, and working experience developing applications with PL/SQL. Required prerequisites are *Oracle Database 10g: Develop PL/SQL Program Units* or *Oracle Database 10g: Program with PL/SQL* .

How This Course Is Organized

Oracle Database 10g: Advanced PL/SQL is an instructor-led course featuring lectures and hands-on exercises. Online demonstrations and practice sessions reinforce the concepts and skills introduced.

Related Publications

Oracle Publications

Title	Part Number
Oracle Database Concepts 10g Release 1 (10.1)	B10743-01
Oracle Database SQL Reference Release 1 (10.1)	B10759-01
PL/SQL Packages and Types Reference 10g Release 1 (10.1)	B10802-01
PL/SQL User's Guide and Reference 10g Release 1 (10.1)	B10807-01
Oracle Database Application Developer's Guide - Fundamentals 10g Release 1 (10.1)	B10795-01
Oracle Database Application Developer's Guide - Object-Relational Features 10g Release 1 (10.1)	B10799-01
Oracle Database Performance Tuning Guide 10g Release 1 (10.1)	B10752-01

Additional Publications

- System release bulletins
- Installation and User's guides
- *read.me* files
- International Oracle User's Group (IOUG) articles
- *Oracle Magazine*
- OTN *(http://otn.oracle.com/)*

Typographic Conventions

The following are two lists of typographical conventions used specifically within text or within code.

Typographic Conventions within Text

Convention	Object or Term	Example
Uppercase	Commands, functions, column names, PL/SQL objects, schemas	Use the `SELECT` command to view information stored in the `CUST_LAST_NAME` column of the `CUSTOMERS` table.
Lowercase	File names, syntax variables, usernames, passwords	**where:** `role` is the name of the role italic to be created.
Initial cap	Trigger and button names	Assign a `When-Validate-Item` trigger to the `ORD` block. Click Cancel.
Italic	Books, names of courses and manuals, and emphasized words or phrases	For more information on the subject, see *Oracle Database Concepts 10g Release 1*. Do *not* save changes to the database.
Quotation marks	Lesson module titles referenced within a course	This subject is covered in Lesson 7, "Performance and Tuning."

Typographic Conventions (continued)

Typographic Conventions within Code

Convention	Object or Term	Example
Uppercase	Commands, functions	`SELECT customer_id` `FROM    customers;`
Lowercase, italic	Syntax variables	`CREATE ROLE `*`rolename`*`;`

Initial cap	Forms triggers	`Form module: ORD`
		`Trigger level: S_ITEM.QUANTITY`
		`item`
		`Trigger name: When-Validate-Item`
		`. . .`

Lowercase	Column names, table names, filenames,	`. . .`
		`SELECT cust_last_name, cust_email`
		`FROM    customers;`

	PL/SQL objects	`EXECUTE dbms_output.put_line('a')`
Bold	Text that must be entered by a user	**`SQLDBA>  CREATE USER oe`** **`     2>  IDENTIFIED BY oe;`**

I

Introduction

Course Objectives

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

- **Design PL/SQL packages and program units that execute efficiently**
- **Write code to interface with external applications and the operating system**
- **Create PL/SQL applications that use collections**
- **Write and tune PL/SQL code effectively to maximize performance**
- **Implement a virtual private database with fine-grained access control**
- **Perform code analysis to find program ambiguities, and test, trace, and profile PL/SQL code**

ORACLE

Course Objectives

In this course, you learn how to use the advanced features of PL/SQL in order to design and tune PL/SQL to interface with the database and other applications in the most efficient manner. Using advanced features of program design, packages, cursors, extended interface methods, and collections, you learn how to write powerful PL/SQL programs. Programming efficiency, use of external C and Java routines, PL/SQL server pages, and fine-grained access are covered in this course.

Oracle Complete Solution

Clients

Any browser   Any mail client   Any FTP client

System management

Internet applications

Business logic and data   Presentation and business logic

Databases   Application servers

Network services

Development tools

SQL

PL/SQL

Java
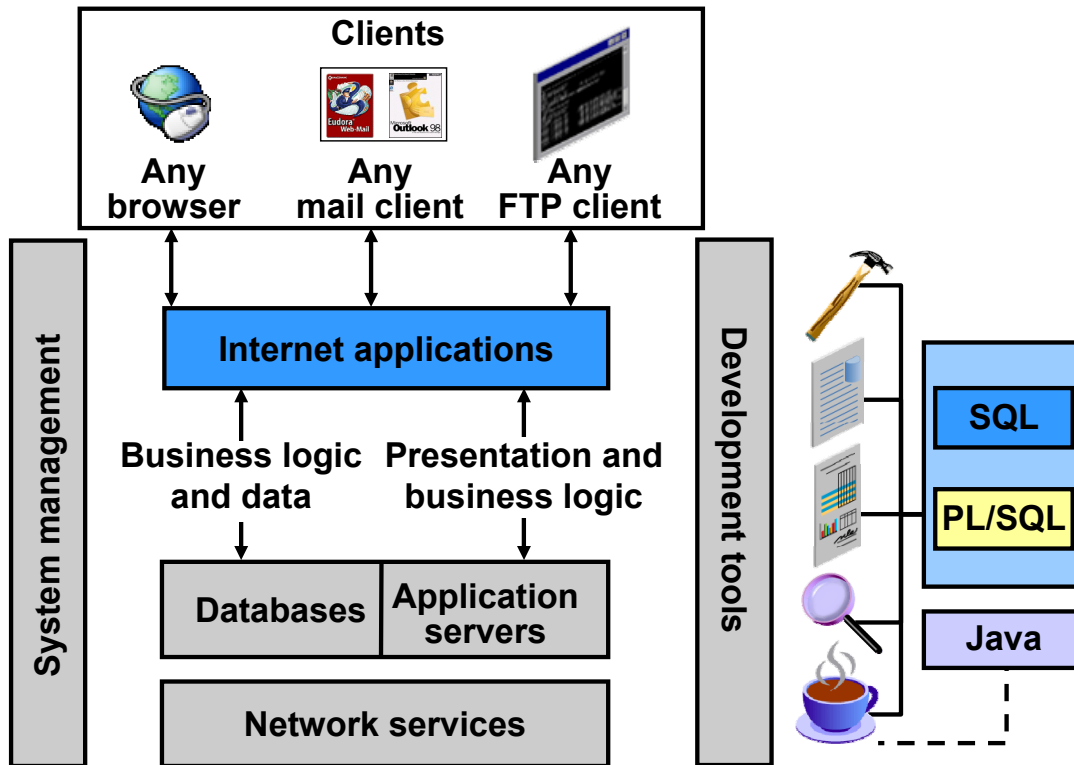
ORACLE

Oracle Complete Solution

The Oracle Internet Platform is built on three core components:
- Browser-based clients to process presentation
- Application servers to execute business logic and serve presentation logic to browser-based clients
- Databases to execute database-intensive business logic and serve data

Oracle offers a wide variety of the most advanced graphical user interface (GUI)–driven development tools to build business applications, as well as a large suite of software applications for many areas of business and industry. Stored procedures, functions, and packages can be written by using SQL, PL/SQL, Java, or XML. This course concentrates on the advanced features of PL/SQL.

Course Agenda

Day 1
- **PL/SQL Programming Concepts Review**
- **Design Considerations**
- **Collections**
- **Advanced Interface Methods**

Day 2
- **PL/SQL Server Pages**
- **Fine-Grained Access Control**
- **Performance and Tuning**
- **Analyzing PL/SQL Code**

ORACLE

Agenda

In this two-day course, you start with a review of PL/SQL concepts before progressing into the new and advanced topics. By the end of day one, you should have covered design considerations for your program units, how to use collections effectively, and how to call C and Java code from your PL/SQL programs.

On day two, you learn how to create and deploy a PL/SQL server page on a browser, how to implement security through packages, how to analyze and identify performance issues, and how to tune your programs.

Tables Used in This Course

- **Sample schemas used are:**
  - `Order Entry (OE)` **schema**
  - `Human Resources (HR)` **schema**
- **Primarily use the `OE` schema.**
- **The `OE` schema can view the `HR` tables.**
- **Appendix B contains more information about the sample schemas.**

Tables Used in This Course

The sample company portrayed by Oracle Database Sample Schemas operates worldwide to fulfil orders for several different products. The company has several divisions:

- The Human Resources division tracks information about the employees and facilities of the company.
- The Order Entry division tracks product inventories and sales of the company's products through various channels.
- The Sales History division tracks business statistics to facilitate business decisions.

Each of these divisions is represented by a schema.

This course primarily uses the `Order Entry (OE)` sample schema.

Note: More details about the sample schema are found in Appendix B.

All scripts necessary to create the `OE` schema reside in the `$ORACLE_HOME/demo/schema/order_entry` folder.

All scripts necessary to create the `HR` schema reside in the `$ORACLE_HOME/demo/schema/human_resources` folder.

The `Order Entry` Schema
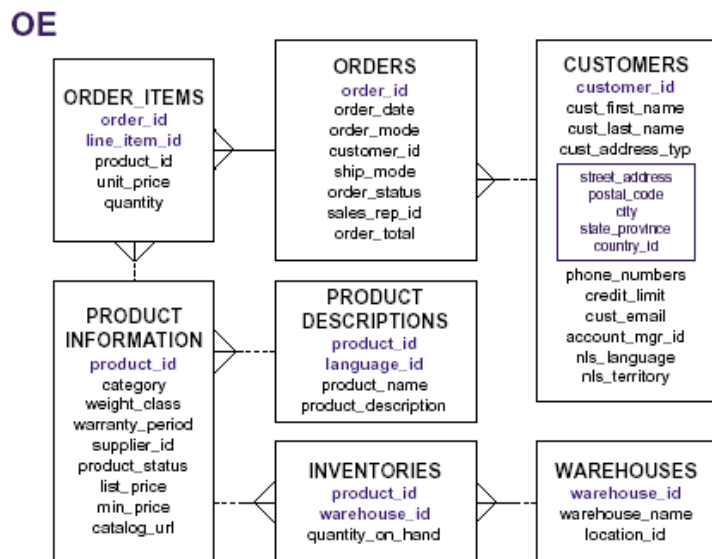
The `Order Entry` (`OE`) Schema

The company sells several categories of products, including computer hardware and software, music, clothing, and tools. The company maintains product information that includes product identification numbers, the category into which the product falls, the weight group (for shipping purposes), the warranty period if applicable, the supplier, the status of the product, a list price, a minimum price at which a product will be sold, and a URL address for manufacturer information.

Inventory information is also recorded for all products, including the warehouse where the product is available and the quantity on hand. Because products are sold worldwide, the company maintains the names of the products and their descriptions in several different languages.

The company maintains warehouses in several locations to facilitate filling customer orders. Each warehouse has a warehouse identification number, name, and location identification number.

The `Order Entry` (`OE`) Schema (continued)

Customer information is tracked in some detail. Each customer is assigned an identification number. Customer records include name, street address, city or province, country, phone numbers (up to five phone numbers for each customer), and postal code. Some customers order through the Internet, so e-mail addresses are also recorded. Because of language differences among customers, the company records the NLS language and territory of each customer. The company places a credit limit on its customers to limit the amount they can purchase at one time. Some customers have account managers, whom we monitor. We keep track of a customer's phone number. At present, we do not know how many phone numbers a customer might have, but we try to keep track of all of them. Because of the language differences of our customers, we identify the language and territory of each customer.

When a customer places an order, the company tracks the date of the order, the mode of the order, status, shipping mode, total amount of the order, and the sales representative who helped place the order. This may be the same individual as the account manager for a customer, it may be someone else, or, in the case of an order over the Internet, the sales representative is not recorded. In addition to the order information, the company also tracks the number of items ordered, the unit price, and the products ordered.

For each country in which it does business, the company records the country name, currency symbol, currency name, and the region where the country resides geographically. This data is useful to interact with customers living in different geographic regions around the world.

The Human Resources Schema
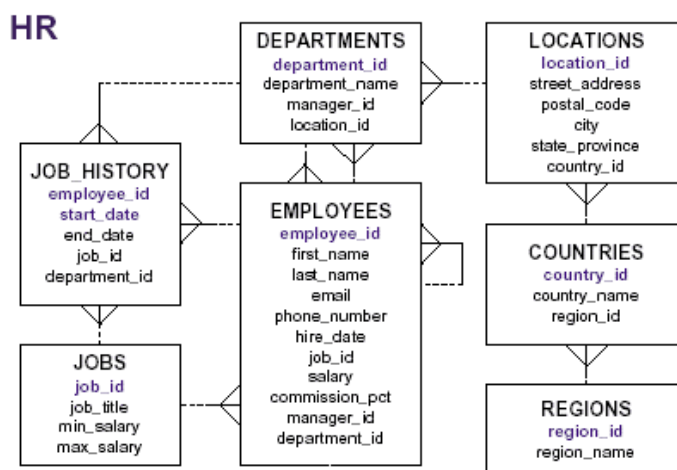
The Human Resources (HR) Schema

In the human resources records, each employee has an identification number, e-mail address, job identification code, salary, and manager. Some employees earn a commission in addition to their salary.

The company also tracks information about jobs within the organization. Each job has an identification code, job title, and a minimum and maximum salary range for the job. Some employees have been with the company for a long time and have held different positions within the company. When an employee switches jobs, the company records the start date and end date of the former job, the job identification number, and the department.

The sample company is regionally diverse, so it tracks the locations of not only its warehouses but also its departments. Each company employee is assigned to a department. Each department is identified by a unique department number and a short name. Each department is associated with one location. Each location has a full address that includes the street address, postal code, city, state or province, and country code.

For each location where it has facilities, the company records the country name, currency symbol, currency name, and the region where the country resides geographically.

Oracle University and En-Sof Informatica E Treinamento Ltda use only

PL/SQL Programming Concepts: Review

Objectives

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

- **Identify PL/SQL block structure**
- **Describe PL/SQL basics**
- **List restrictions and guidelines on calling functions from SQL expressions**
- **Identify how explicit cursors are processed**
- **Handle exceptions**
- **Use the `raise_application_error` procedure**
- **Manage dependencies**

Objectives

PL/SQL supports various programming constructs. This lesson reviews the basic concept of PL/SQL programming. This lesson also reviews how to:

- Create subprograms
- Use cursors
- Handle exceptions
- Identify predefined Oracle server errors
- Manage dependencies

A quiz at the end of the lesson will assess your knowledge of PL/SQL.

Note: The quiz is optional. Solutions to the quiz are provided in Appendix A.

PL/SQL Block Structure
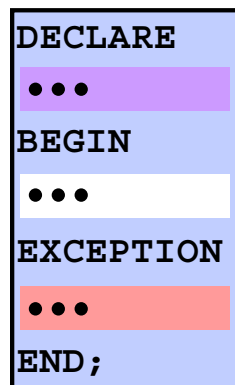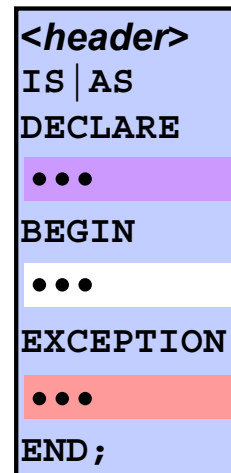
```
DECLARE
 ● ● ●

BEGIN
 ● ● ●

EXCEPTION
 ● ● ●

END;
```

**Anonymous
PL/SQL block**

```
<header>
IS|AS
DECLARE
 ● ● ●

BEGIN
 ● ● ●

EXCEPTION
 ● ● ●

END;
```

**Stored
program unit**

PL/SQL Block Structure

An anonymous PL/SQL block structure consists of an optional DECLARE section, a mandatory BEGIN-END block, and an optional EXCEPTION section before the END statement of the main block.

A stored program unit has a mandatory header section. This section defines whether the program unit is a function, procedure, or a package. A stored program unit also has other sections mentioned for the anonymous PL/SQL block.

Every PL/SQL construct is made from one or more blocks. These blocks can be entirely separate, or nested within one another. Therefore, one block can represent a small part of another block, which in turn can be part of the whole unit of code.

Naming Conventions

Advantages of proper naming conventions:

- **Easier to read**
- **Understandable**
- **Give information about the functionality**
- **Easier to debug**
- **Ensure consistency**

ORACLE

Naming Conventions

A proper naming convention makes the code easier to read and more understandable. It helps you understand the functionality of the identifier. If the code is written using proper naming conventions, you can easily find an error and rectify it. Most importantly, it ensures consistency among the code written by different developers.

The following table shows the naming conventions followed in this course:

Identifier	Convention	Example
Variable	v_prefix	v_product_name
Constant	c_prefix	c_tax
Parameter	p_prefix	p_cust_id
Exception	e_prefix	e_check_credit_limit
Cursor	cur_prefix	cur_orders
Type	typ_prefix	typ_customer

Procedures

A procedure is:

- **A named PL/SQL block that performs a sequence of actions**
- **Stored in the database as a schema object**
- **Used to promote reusability and maintainability**

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

ORACLE

Procedures

A procedure is a named PL/SQL block that can accept parameters (sometimes referred to as arguments). Generally, you use a procedure to perform an action. A procedure is compiled and stored in the database as a schema object. Procedures promote reusability and maintainability.

Parameters are used to transfer data values to and from the calling environment and the procedure (or subprogram). Parameters are declared in the subprogram header, after the name and before the declaration section for local variables.

Parameters are subject to one of the three parameter-passing modes: IN, OUT, or IN OUT.
- An IN parameter passes a constant value from the calling environment into the procedure.
- An OUT parameter passes a value from the procedure to the calling environment.
- An IN OUT parameter passes a value from the calling environment to the procedure and a possibly different value from the procedure back to the calling environment using the same parameter.

Functions

A function is:

- **A block that returns a value**
- **Stored in the database as a schema object**
- **Called as part of an expression or used to provide a parameter value**

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

Functions

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

Functions can be stored in the database as schema objects for repeated execution. A function that is stored in the database is referred to as a stored function. Functions can also be created on client-side applications.

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

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

Function: Example

- **Create the function:**

```
CREATE OR REPLACE FUNCTION get_credit
 (v_id customers.customer_id%TYPE) RETURN NUMBER IS
  v_credit customers.credit_limit%TYPE := 0;
BEGIN
  SELECT credit_limit
  INTO   v_credit
  FROM   customers
  WHERE  customer_id = v_id;
  RETURN v_credit;
END get_credit;
/
```

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

```
EXECUTE dbms_output.put_line(get_credit(101))
```

ORACLE

Function: Example

The get_credit function is created with a single input parameter and returns the credit limit as a number.

The get_credit function follows the common programming practice of assigning a returning value to a local variable and uses a single RETURN statement in the executable section of the code to return the value stored in the local variable. If your function has an exception section, then it may also contain a RETURN statement.

Invoke a function as part of a PL/SQL expression, because the function will return a value to the calling environment. The second code box uses the SQL*Plus EXECUTE command to call the DBMS_OUTPUT.PUT_LINE procedure whose argument is the return value from the get_credit function. In this case, get_credit is invoked first to calculate the credit limit of the customer with ID 101. The credit_limit value returned is supplied as the value of the DBMS_OUTPUT.PUT_LINE parameter, which displays the result (if you have executed a SET SERVEROUTPUT ON).

Note: A function must always return a value. The example does not return a value if a row is not found for a given ID. Ideally, create an exception handler to return a value as well.

Ways to Execute Functions

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

```
VARIABLE v_credit NUMBER
EXECUTE :v_credit := get_credit(101)
```

  - **Using a local variable to obtain the result:**

```
DECLARE v_credit customers.credit_limit%type;
BEGIN
  v_credit := get_credit(101); ...
END;
```

- **Use as a parameter to another subprogram**

```
EXECUTE dbms_output.put_line(get_credit(101))
```

- **Use in a SQL statement (subject to restrictions)**

```
SELECT get_credit(customer_id) FROM customers;
```

Ways to Execute Functions

If functions are designed thoughtfully, they can be powerful constructs. Functions can be invoked in the following ways:

- **As part of PL/SQL expressions:** You can use host or local variables to hold the returned value from a function. The first example in the slide uses a host variable and the second example uses a local variable in an anonymous block.
- **As a parameter to another subprogram:** The third example in the slide demonstrates this usage. The get_credit function, with all its arguments, is nested in the parameter required by the DBMS_OUTPUT.PUT_LINE procedure. This comes from the concept of nesting functions as discussed in the *Oracle Database 10g: SQL Fundamentals I* course.
- **As an expression in a SQL statement:** The last example shows how a function can be used as a single-row function in a SQL statement.

Note: The restrictions and guidelines that apply to functions when used in a SQL statement are discussed in the next few pages.

Restrictions on Calling Functions from SQL Expressions

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

Restrictions on Calling Functions from SQL Expressions

The user-defined PL/SQL functions that are callable from SQL expressions must meet the following requirements:
- The function must be stored in the database.
- The function parameters must be input parameters and should be valid SQL data types.
- The functions must return data types that are valid SQL data types. They cannot be PL/SQL-specific data types such as `BOOLEAN`, `RECORD`, or `TABLE`. The same restriction applies to the parameters of the function.

The following restrictions apply when calling a function in a SQL statement:
- Parameters must use positional notation. Named notation is not supported.
- You must own or have the `EXECUTE` privilege on the function.

Other restrictions on a user-defined function include the following:
- It cannot be called from the `CHECK` constraint clause of a `CREATE TABLE` or `ALTER TABLE` statement.
- It cannot be used to specify a default value for a column.

Note: Only stored functions are callable from SQL statements. Stored procedures cannot be called unless invoked from a function that meets the preceding requirements.

Guidelines for Calling Functions from SQL Expressions

Functions called from:

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

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

Guidelines for Calling Functions from SQL Expressions

To execute a SQL statement that calls a stored function, the Oracle server must know whether the function is free of specific side effects. The side effects are unacceptable changes to database tables.

Additional restrictions apply when a function is called in expressions of SQL statements. In particular, when a function is called from:
- A `SELECT` statement or a parallel `UPDATE` or `DELETE` statement, the function cannot modify any database table
- An `UPDATE` or `DELETE` statement, the function cannot query or modify any database table modified by that statement
- A `SELECT`, `INSERT`, `UPDATE`, or `DELETE` statement, the function cannot execute directly or indirectly through another subprogram, a SQL transaction control statement such as:
  - A `COMMIT` or `ROLLBACK` statement
  - A session control statement (such as `SET ROLE`)
  - A system control statement (such as `ALTER SYSTEM`)
  - Any DDL statements (such as `CREATE`), because they are followed by an automatic commit

PL/SQL Packages: Review

PL/SQL packages:
- **Group logically related components:**
  - **PL/SQL types**
  - **Variables, data structures, and exceptions**
  - **Subprograms: procedures and functions**
- **Consist of two parts:**
  - **A specification**
  - **A body**
- **Enable the Oracle server to read multiple objects into memory simultaneously**

PL/SQL Packages: Review

PL/SQL packages enable you to bundle related PL/SQL types, variables, data structures, exceptions, and subprograms into one container. For example, an Order Entry package can contain procedures for adding and deleting customers and orders, functions for calculating annual sales, and credit limit variables.

A package usually consists of two parts stored separately in the database:
- A specification
- A body (optional)

The package itself cannot be called, parameterized, or nested. After writing and compiling, the contents can be shared with many applications.

When a PL/SQL-packaged construct is referenced for the first time, the whole package is loaded into memory. Subsequent access to constructs in the same package does not require disk input/output (I/O).

Components of a PL/SQL Package
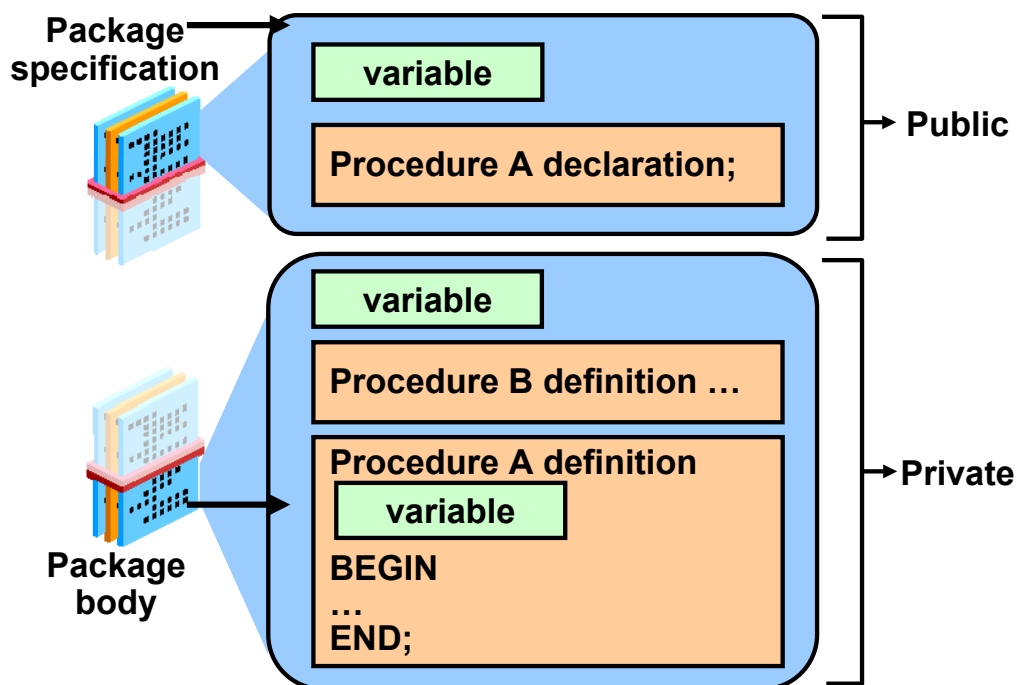
Components of a PL/SQL Package

You create a package in two parts:

- The *package specification* is the interface to your applications. It declares the public types, variables, constants, exceptions, cursors, and subprograms available for use. The package specification may also include pragmas, which are directives to the compiler.
- The *package body* defines its own subprograms and must fully implement subprograms declared in the specification part. The package body may also define PL/SQL constructs, such as types variables, constants, exceptions, and cursors.

Public components are declared in the package specification. The specification defines a public application programming interface (API) for users of package features and functionality. That is, public components can be referenced from any Oracle server environment that is external to the package.

Private components are placed in the package body and can be referenced only by other constructs within the same package body. Private components can reference the public components of the package.

Note: If a package specification does not contain subprogram declarations, then there is no requirement for a package body.

Oracle University and En-Sof Informatica E Treinamento Ltda use only

Creating the Package Specification

Syntax:

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

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

ORACLE

Creating the Package Specification

- To create packages, you declare all public constructs within the package specification.
  - Specify the OR REPLACE option, if overwriting an existing package specification.
  - Initialize a variable with a constant value or formula within the declaration, if required; otherwise, the variable is initialized implicitly to NULL.
- The following are definitions of items in the package syntax:
  - *package_name* specifies a name for the package that must be unique among objects within the owning schema. Including the package name after the END keyword is optional.
  - *public type and variable declarations* declares public variables, constants, cursors, exceptions, user-defined types, and subtypes.
  - *subprogram specifications* specifies the public procedure or function declarations.

Note: The package specification should contain procedure and function headings terminated by a semicolon, without the IS (or AS) keyword and its PL/SQL block. The implementation of a procedure or function that is declared in a package specification is done in the package body.

Creating the Package Body

Syntax:

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

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

ORACLE

Creating the Package Body

Create a package body to define and implement all public subprograms and supporting private constructs. When creating a package body, do the following:

- Specify the OR REPLACE option to overwrite an existing package body.
- Define the subprograms in an appropriate order. The basic principle is that you must declare a variable or subprogram before it can be referenced by other components in the same package body. It is common to see all private variables and subprograms defined first and the public subprograms defined last in the package body.
- The package body must complete the implementation for all procedures or functions declared in the package specification.

The following are definitions of items in the package body syntax:

- **package_name** specifies a name for the package that must be the same as its package specification. Using the package name after the END keyword is optional.
- **private type and variable declarations** declares private variables, constants, cursors, exceptions, user-defined types, and subtypes.
- **subprogram bodies** specifies the full implementation of any private and/or public procedures or functions.
- **[BEGIN initialization statements]** is an optional block of initialization code that executes when the package is first referenced.

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**

Oracle University and En-Sof Informatica E Treinamento Ltda use only

Cursor

You have already learned that you can include SQL statements that return a single row in a PL/SQL block. The data retrieved by the SQL statement should be held in variables using the INTO clause.

Where Does Oracle Process SQL Statements?

The Oracle server allocates a private memory area called the context area for processing SQL statements. The SQL statement is parsed and processed in this area. Information required for processing and information retrieved after processing are all stored in this area. Because this area is internally managed by the Oracle server, you have no control over this area. A cursor is a pointer to the context area. However, this cursor is an implicit cursor and is automatically managed by the Oracle server. When the executable block contains a SQL statement, an implicit cursor is created.

There are two types of cursors:
- **Implicit cursors:** Implicit cursors are created and managed by the Oracle server. You do not have access to them. The Oracle server creates such a cursor when it has to execute a SQL statement.

Cursor (continued)

- **Explicit cursors:** As a programmer, you may want to retrieve multiple rows from a database table, have a pointer to each row that is retrieved, and work on the rows one at a time. In such cases, you can declare cursors explicitly depending on your business requirements. Such cursors that are declared by programmers are called explicit cursors. You declare these cursors in the declarative section of a PL/SQL block. Remember that you can also declare variables and exceptions in the declarative section.

Processing Explicit Cursors

The following three commands are used to process an explicit cursor:

- **OPEN**
- **FETCH**
- **CLOSE**

Alternatively, you can also use a cursor FOR loops.

Processing Explicit Cursors

You declare an explicit cursor when you need exact control over query processing. You use three commands to control a cursor:

- OPEN
- FETCH
- CLOSE

You initialize the cursor with the OPEN command, which recognizes the result set. Then you execute the FETCH command repeatedly in a loop until all rows have been retrieved. Alternatively, you can use a BULK COLLECT clause to fetch all rows at once. After the last row has been processed, you release the cursor by using the CLOSE command.

Explicit Cursor Attributes

Every explicit cursor has the following four attributes:

- `cursor_name%FOUND`
- `cursor_name%ISOPEN`
- `cursor_name%NOTFOUND`
- `cursor_name%ROWCOUNT`

Cursor Attributes

When cursor attributes are appended to the cursors, they return useful information regarding the execution of the DML statement. The following are the four cursor attributes:

- `cursor_name%FOUND`: Returns TRUE if the last fetch returned a row. Returns NULL before the first fetch from an OPEN cursor. Returns FALSE if the last fetch failed to return a row.
- `cursor_name%ISOPEN`: Returns TRUE if the cursor is open, otherwise returns FALSE.
- `cursor_name%NOTFOUND`: Returns FALSE if the last fetch returned a row. Returns NULL before the first fetch from an OPEN cursor. Returns TRUE if the last fetch failed to return a row.
- `cursor_name%ROWCOUNT`: Returns zero before the first fetch. After every fetch returns the number of rows fetched so far.

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.**

ORACLE

Cursor FOR Loops

A cursor FOR loop processes rows in an explicit cursor. It is a shortcut because the cursor is opened, a row is fetched once for each iteration in the loop, the loop exits when the last row is processed, and the cursor is closed automatically. The loop itself is terminated automatically at the end of the iteration where the last row is fetched.

In the syntax:

record_name	Is the name of the implicitly declared record
cursor_name	Is a PL/SQL identifier for the previously declared cursor

Guidelines

- Do not declare the record that controls the loop because it is declared implicitly.
- Test the cursor attributes during the loop, if required.
- Supply the parameters for a cursor, if required, in parentheses following the cursor name in the FOR statement.

Cursor: Example

```
SET SERVEROUTPUT ON
DECLARE
  CURSOR cur_cust IS
    SELECT cust_first_name, credit_limit
    FROM customers
    WHERE credit_limit > 4000;
BEGIN
  FOR v_cust_record IN cur_cust
   LOOP
    DBMS_OUTPUT.PUT_LINE
     (v_cust_record.cust_first_name ||' '||
      v_cust_record.credit_limit);
   END LOOP;
END;
/
```

Cursor: Example

The example shows the use of a cursor `FOR` loop.

The `cust_record` is the record that is implicitly declared. You can access the fetched data with this implicit record as shown in the slide. Note that no variables are declared to hold the fetched data using the `INTO` clause. The code does not have `OPEN` and `CLOSE` statements to open and close the cursor, respectively.

Handling Exceptions

- **An exception is an error in 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

An exception is an error in PL/SQL that is raised during the execution of a block. A block always terminates when PL/SQL raises an exception, but you can specify an exception handler to perform final actions before the block ends.

Methods for Raising an Exception

- An Oracle error occurs and the associated exception is raised automatically. For example, if the error ORA-01403 occurs when no rows are retrieved from the database in a SELECT statement, then PL/SQL raises the NO_DATA_FOUND exception. These errors are converted into predefined exceptions.
- Depending on the business functionality your program is implementing, you may have to explicitly raise an exception. You raise an exception explicitly by issuing the RAISE statement within the block. The exception being raised may be either user-defined or predefined.
- There are some non-predefined Oracle errors. These errors are any standard Oracle errors that are not predefined. You can explicitly declare exceptions and associate them with the non-predefined Oracle errors.

Handling Exceptions

Exception raised

Is the exception trapped?

no → **Terminate abruptly.** → **Propagate the exception.**

yes → **Execute statements in the EXCEPTION section.** → **Terminate gracefully.**
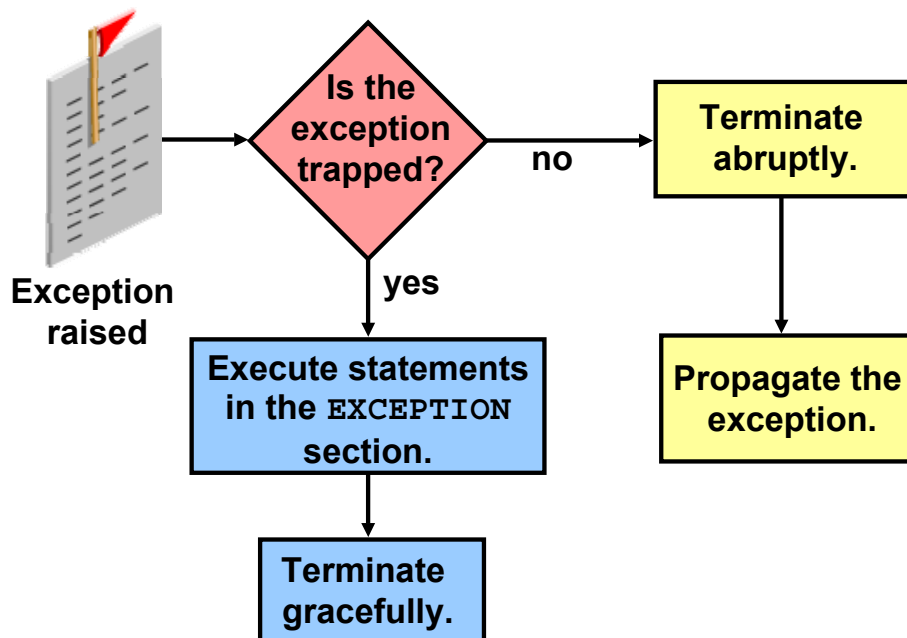
Handling Exceptions (continued)

Trapping an Exception

Include an EXCEPTION section in your PL/SQL program to trap exceptions. If the exception is raised in the executable section of the block, processing branches to the corresponding exception handler in the exception section of the block. If PL/SQL successfully handles the exception, then the exception does not propagate to the enclosing block or to the calling environment. The PL/SQL block terminates successfully.

Propagating an Exception

If the exception is raised in the executable section of the block and there is no corresponding exception handler, the PL/SQL block terminates with failure and the exception is propagated to an enclosing block or to the calling environment. The calling environment can be any application, such as SQL*Plus, that invokes the PL/SQL program.

Exceptions: Example

```
DECLARE
   v_lname VARCHAR2(15);
BEGIN
   SELECT cust_last_name INTO v_lname FROM customers
   WHERE cust_first_name='Ally';
   DBMS_OUTPUT.PUT_LINE ('Ally''s last name is : '
                           ||v_lname);
EXCEPTION
   WHEN TOO_MANY_ROWS THEN
   DBMS_OUTPUT.PUT_LINE (' Your select statement
   retrieved multiple rows. Consider using a
   cursor.');
END;
/
```

Exceptions: Example

You have written PL/SQL blocks with a declarative section (beginning with the keyword
DECLARE) and an executable section (beginning and ending with the keywords BEGIN and
END, respectively). For exception handling, include another optional section called the
EXCEPTION section. This section begins with the keyword EXCEPTION. If present, this is the
last section in a PL/SQL block. Examine the code in the slide to see the EXCEPTION section.

The output of this code is shown below:

Your select statement retrieved multiple rows. Consider using a
cursor.

PL/SQL procedure successfully completed.

When the exception is raised, the control shifts to the EXCEPTION section and all the statements
in the EXCEPTION section are executed. The PL/SQL block terminates with normal, successful
completion.

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`

Predefined Oracle Server Errors

You can reference predefined Oracle server errors by using its predefined name within the corresponding exception-handling routine.

For a complete list of predefined exceptions, see *PL/SQL User's Guide and Reference*.

Note: PL/SQL declares predefined exceptions in the STANDARD package.

Predefined Oracle Server Errors (continued)

Exception Name	Oracle Server Error Number	Description
ACCESS_INTO_NULL	ORA-06530	Attempted to assign values to the attributes of an uninitialized object
CASE_NOT_FOUND	ORA-06592	None of the choices in the WHEN clauses of a CASE statement is selected, and there is no ELSE clause.
COLLECTION_IS_NULL	ORA-06531	Attempted to apply collection methods other than EXISTS to an uninitialized nested table or varray
CURSOR_ALREADY_OPEN	ORA-06511	Attempted to open an already open cursor
DUP_VAL_ON_INDEX	ORA-00001	Attempted to insert a duplicate value
INVALID_CURSOR	ORA-01001	Illegal cursor operation occurred
INVALID_NUMBER	ORA-01722	Conversion of character string to number fails
LOGIN_DENIED	ORA-01017	Logging on to the Oracle server with an invalid username or password
NO_DATA_FOUND	ORA-01403	Single-row SELECT returned no data
NOT_LOGGED_ON	ORA-01012	PL/SQL program issues a database call without being connected to the Oracle server
PROGRAM_ERROR	ORA-06501	PL/SQL has an internal problem
ROWTYPE_MISMATCH	ORA-06504	Host cursor variable and PL/SQL cursor variable involved in an assignment have incompatible return types

Predefined Oracle Server Errors (continued)

Exception Name	Oracle Server Error Number	Description
STORAGE_ERROR	ORA-06500	PL/SQL ran out of memory or memory is corrupted.
SUBSCRIPT_BEYOND_COUNT	ORA-06533	Referenced a nested table or varray element by using an index number larger than the number of elements in the collection
SUBSCRIPT_OUTSIDE_LIMIT	ORA-06532	Referenced a nested table or varray element by using an index number that is outside the legal range (for example −1)
SYS_INVALID_ROWID	ORA-01410	The conversion of a character string into a universal ROWID fails because the character string does not represent a valid ROWID.
TIMEOUT_ON_RESOURCE	ORA-00051	Time-out occurred while the Oracle server was waiting for a resource.
TOO_MANY_ROWS	ORA-01422	Single-row SELECT returned more than one row.
VALUE_ERROR	ORA-06502	Arithmetic, conversion, truncation, or size-constraint error occurred.
ZERO_DIVIDE	ORA-01476	Attempted to divide by zero

Trapping Non-Predefined Oracle Server Errors
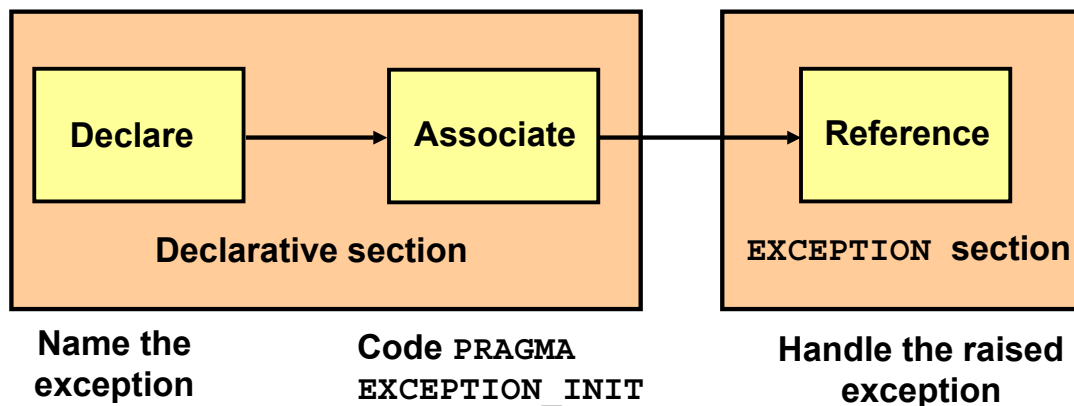
Declare → **Associate** → **Reference**

Declarative section   **EXCEPTION section**

Name the exception   **Code PRAGMA EXCEPTION_INIT**   **Handle the raised exception**

Trapping Non-Predefined Oracle Server Errors

Non-predefined exceptions are similar to predefined exceptions; however, they are not defined as PL/SQL exceptions in the Oracle server. They are standard Oracle errors. You can create exceptions with standard Oracle errors by using the PRAGMA EXCEPTION_INIT function. Such exceptions are called non-predefined exceptions.

You can trap a non-predefined Oracle server error by declaring it first. The declared exception is raised implicitly. In PL/SQL, PRAGMA EXCEPTION_INIT instructs the compiler to associate an exception name with an Oracle error number. That allows you to refer to any internal exception by name and to write a specific handler for it.

Note: PRAGMA (also called pseudoinstructions) is the keyword that signifies that the statement is a compiler directive, which is not processed when the PL/SQL block is executed. Rather, it directs the PL/SQL compiler to interpret all occurrences of the exception name within the block as the associated Oracle server error number.

Trapping User-Defined Exceptions

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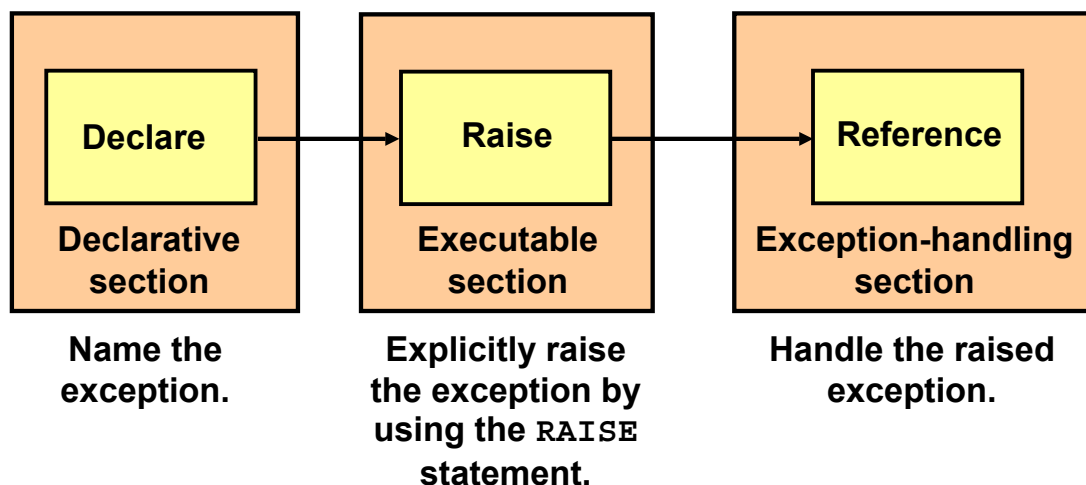
Trapping User-Defined Exceptions

With PL/SQL, you can define your own exceptions. You define exceptions depending on the requirements of your application. For example, you may prompt the user to enter a department number.

Define an exception to deal with error conditions in the input data. Check whether the department number exists. If it does not, then you may have to raise the user-defined exception. PL/SQL exceptions must be:

- Declared in the declarative section of a PL/SQL block
- Raised explicitly with RAISE statements
- Handled in the EXCEPTION section

The `RAISE_APPLICATION_ERROR` Procedure

Syntax:

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

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

The `RAISE_APPLICATION_ERROR` Procedure

Use the `RAISE_APPLICATION_ERROR` procedure to communicate a predefined exception interactively by returning a nonstandard error code and error message. With `RAISE_APPLICATION_ERROR`, you can report errors to your application and avoid returning unhandled exceptions.

In the syntax:

`error_number`	Is a user-specified number for the exception between –20,000 and –20,999
`message`	Is the user-specified message for the exception. It is a character string up to 2,048 bytes long.
`TRUE │ FALSE`	Is an optional Boolean parameter. (If `TRUE`, the error is placed on the stack of previous errors. If `FALSE`, the default, the error replaces all previous errors.)

The `RAISE_APPLICATION_ERROR` Procedure

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

The `RAISE_APPLICATION_ERROR` Procedure (continued)

The `RAISE_APPLICATION_ERROR` can be used in either the executable section or the exception section of a PL/SQL program, or both. The returned error is consistent with how the Oracle server produces a predefined, non-predefined, or user-defined error. The error number and message are displayed to the user.

Dependencies

Dependent objects

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

Referenced objects

| Function |
| Package specification |
| Procedure |
| Sequence |
| Synonym |
| Table |
| View |
| User-defined object and collection types |
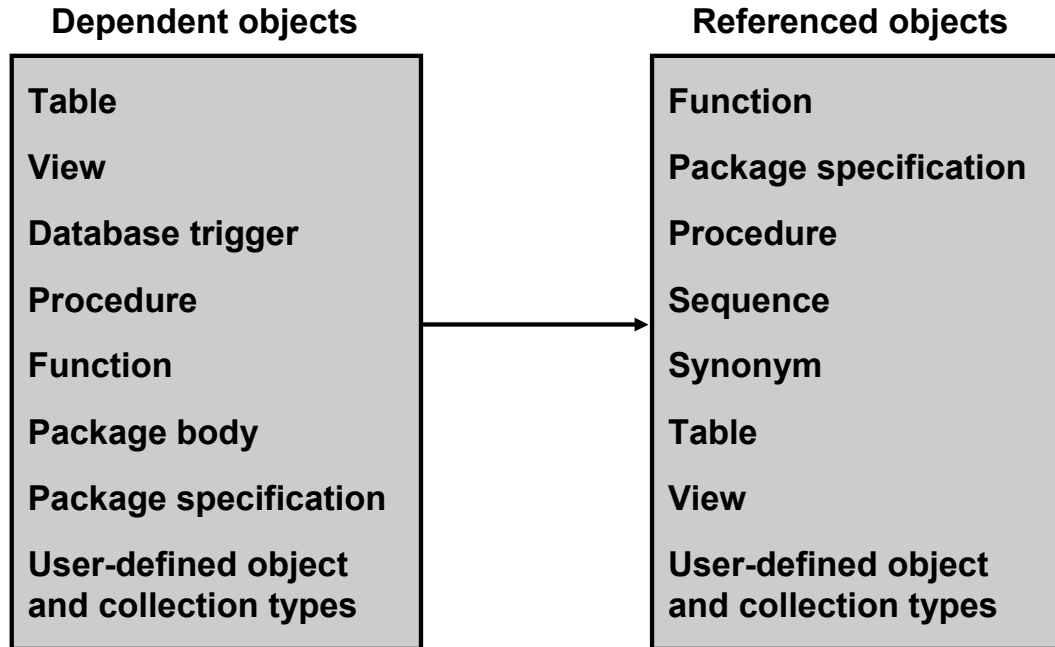
Dependent and Referenced Objects

Some objects reference other objects as part of their definitions. For example, a stored procedure could contain a SELECT statement that selects columns from a table. For this reason, the stored procedure is called a dependent object, whereas the table is called a referenced object.

Dependency Issues

If you alter the definition of a referenced object, dependent objects may or may not continue to work properly. For example, if the table definition is changed, procedure may or may not continue to work without an error.

The Oracle server automatically records dependencies among objects. To manage dependencies, all schema objects have a status (valid or invalid) that is recorded in the data dictionary, and you can view the status in the USER_OBJECTS data dictionary view.

Status	Significance
VALID	The schema object has been compiled and can be immediately used when referenced.
INVALID	The schema object must be compiled before it can be used.

Dependencies

Procedure

xxxxxxxxxxxxxx
vvvvvvvvvvvvvv
xxxxxxxxxxxxxx
vvvvvvvvvvvvvv
xxxxxxxxxxxxxx
vvvvvvvvvvvvvv
xxxxxxxxxxxxxx
vvvvvvvvvvvvvv
xxxxxxxxxxxxxx
vvvvvvvvvvvvvv

View or procedure

Direct dependency

Referenced

Dependent

Direct dependency

Table
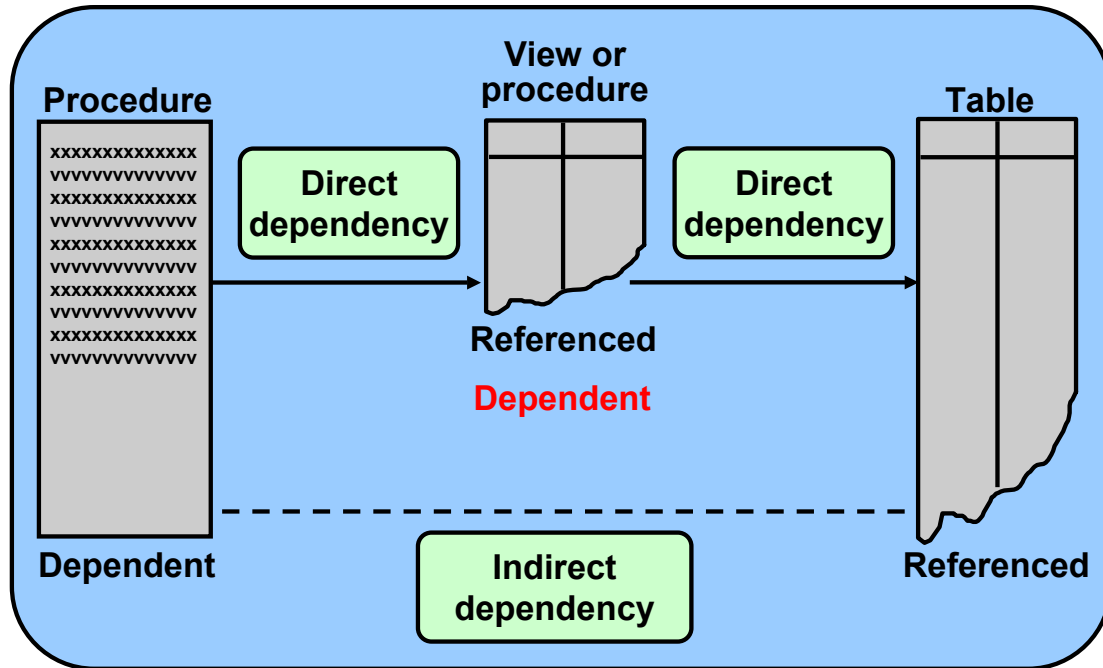
Dependent

Indirect dependency

Referenced

Dependent and Referenced Objects (continued)

A procedure or function can directly or indirectly (through an intermediate view, procedure, function, or packaged procedure or function) reference the following objects:

- Tables
- Views
- Sequences
- Procedures
- Functions
- Packaged procedures or functions

Displaying Direct and Indirect Dependencies

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

```
EXECUTE deptree_fill('TABLE','OE','CUSTOMERS')
```

Displaying Direct and Indirect Dependencies by Using Views

Display direct and indirect dependencies from additional user views called DEPTREE and IDEPTREE; these views are provided by the Oracle database.

Example

1. Make sure that the `utldtree.sql` script has been executed. This script is located in the `$ORACLE_HOME/rdbms/admin` folder.
2. Populate the DEPTREE_TEMPTAB table with information for a particular referenced object by invoking the DEPTREE_FILL procedure. There are three parameters for this procedure:

object_type	Type of the referenced object
object_owner	Schema of the referenced object
object_name	Name of the referenced object

Using Oracle-Supplied Packages

Oracle-supplied packages:

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

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

Using Oracle-Supplied Packages

Packages are provided with the Oracle server to allow either of the following:

- PL/SQL access to certain SQL features
- The extension of the functionality of the database

You can use the functionality provided by these packages when creating your application, or you may simply want to use these packages as ideas when you create your own stored procedures.

Most of the standard packages are created by running catproc.sql.

Some of the Oracle Supplied Packages

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

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

ORACLE

List of Some of the Oracle Supplied Packages

The list of PL/SQL packages provided with an Oracle database grows with the release of new versions. It would be impossible to cover the exhaustive set of packages and their functionality in this course. For more information, refer to *PL/SQL Packages and Types Reference 10g* manual (previously known as the *PL/SQL Supplied Packages Reference)*.

The following is a brief description of some listed packages:

- The DBMS_ALERT package supports asynchronous notification of database events. Messages or alerts are sent on a COMMIT command.
- The DBMS_LOCK package is used to request, convert, and release locks through Oracle Lock Management services.
- The DBMS_SESSION package enables programmatic use of the ALTER SESSION SQL statement and other session-level commands.
- The DBMS_OUTPUT package provides debugging and buffering of text data.
- The HTP package writes HTML-tagged data into database buffers.
- The UTL_FILE package enables reading and writing of operating system text files.
- The UTL_MAIL package enables composing and sending of e-mail messages.
- The DBMS_SCHEDULER package enables scheduling and automated execution of PL/SQL blocks, stored procedures, and external procedures or executables.

DBMS_OUTPUT **Package**

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

- **PUT and PUT_LINE place text in the buffer.**
- **GET_LINE and GET_LINES read the buffer.**
- **Use SET SERVEROUTPUT ON to display messages in SQL*Plus.**
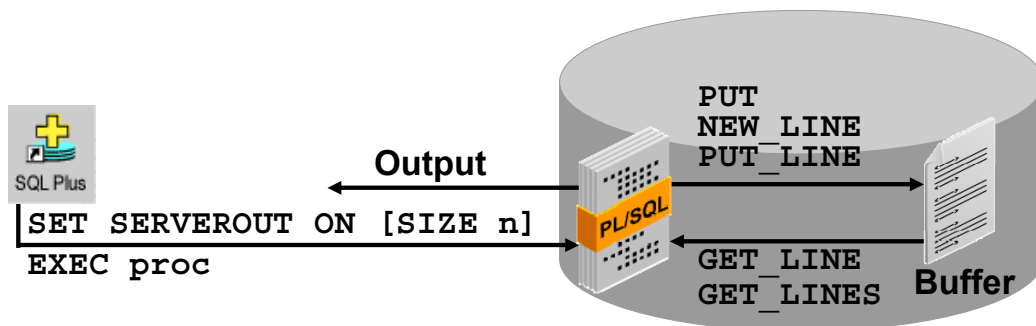

```
SQL Plus                    Output        PUT
                         ←──────────      NEW_LINE
                                          PUT_LINE
                                   PL/SQL  ──────────→
SET SERVEROUT ON [SIZE n] ──────────→
EXEC proc                                 GET_LINE
                                   ←────── GET_LINES    Buffer
```

ORACLE

DBMS_OUTPUT **Package**

The DBMS_OUTPUT package sends textual messages from any PL/SQL block into a buffer in the database. The procedures provided by the package include:

- PUT to append text from the procedure to the current line of the line output buffer
- NEW_LINE to place an end-of-line marker in the output buffer
- PUT_LINE to combine the action of PUT and NEW_LINE; to trim leading spaces
- GET_LINE to retrieve the current line from the buffer into a procedure variable
- GET_LINES to retrieve an array of lines into a procedure-array variable
- ENABLE/DISABLE to enable or disable calls to the DBMS_OUTPUT procedures

The buffer size can be set by using:

- The SIZE *n* option appended to the SET SERVEROUTPUT ON command, where *n* is between 2,000 (the default) and 1,000,000 (1 million characters)
- An integer parameter between 2,000 and 1,000,000 in the ENABLE procedure

Practical Uses

- You can output results to the window for debugging purposes.
- You can trace the code execution path for a function or procedure.
- You can send messages between subprograms and triggers.

Note: There is no mechanism to flush output during the execution of a procedure.

UTL_FILE Package

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

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


```
CREATE DIRECTORY
  my_dir AS '/dir'
```

SQL Plus — EXEC proc → PL/SQL → UTL_FILE ↔ O/S file
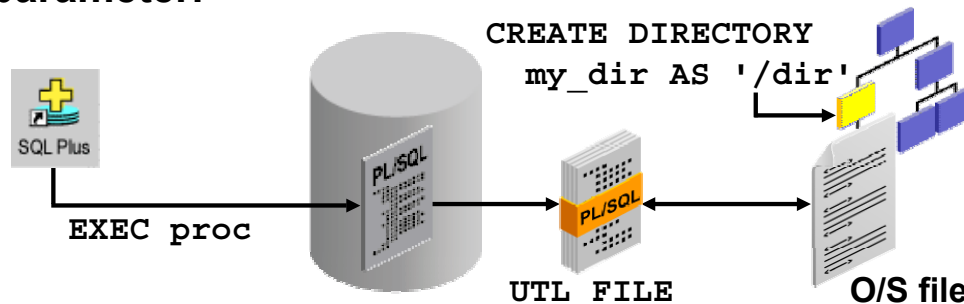
UTL_FILE Package

The Oracle-supplied `UTL_FILE` package is used to access text files in the operating system of the database server. The database provides read and write access to specific operating system directories by using:

- A `CREATE DIRECTORY` statement that associates an alias with an operating system directory. The database directory alias can be granted the `READ` and `WRITE` privileges to control the type of access to files in the operating system. For example:

```
CREATE DIRECTORY my_dir AS '/temp/my_files';
GRANT READ, WRITE ON DIRECTORY my_dir TO public;
```

- The paths specified in the `utl_file_dir` database initialization parameter

The preferred approach is to use the directory alias created by the `CREATE DIRECTORY` statement, which does not require the database to be restarted. The operating system directories specified by using either of these techniques should be accessible to and on the same machine as the database server processes. The path (directory) names may be case sensitive for some operating systems.

Note: The `DBMS_LOB` package can be used to read binary files on the operating system.

Summary

In this lesson, you should have learned how to:

- **Identify a PL/SQL block**
- **Create subprograms**
- **List restrictions and guidelines on calling functions from SQL expressions**
- **Use cursors**
- **Handle exceptions**
- **Use the `raise_application_error` procedure**
- **Identify Oracle-supplied packages**

Summary

This lesson reviewed some basic PL/SQL concepts such as:

- PL/SQL block structure
- Subprograms
- Cursors
- Exceptions
- Oracle-supplied packages

The quiz on the following pages is designed to test and review your PL/SQL knowledge. This knowledge is necessary as a base line for the subsequent chapters to build upon.

Practice 1: PL/SQL Knowledge Quiz

The questions are designed as a refresher. Use the space provided for your answers. If you do not know the answer, go on to the next question. For solutions to this quiz, see Appendix A.

PL/SQL Basics

1. What are the four key areas of the basic `PL/SQL` block? What happens in each area?

2. What is a variable and where is it declared?

3. What is a constant and where is it declared?

4. What are the different modes for parameters and what does each mode do?

5. How does a function differ from a procedure?

6. What are the two main components of a PL/SQL package?

    a. In what order are they defined?

    b. Are both required?

7. How does the syntax of a `SELECT` statement used within a PL/SQL block differ from a `SELECT` statement issued in SQL*Plus?

8. What is a record?

9. What is an `index-by` table?

10. How are loops implemented in PL/SQL?

11. How is branching logic implemented in PL/SQL?

Practice 1: PL/SQL Knowledge Quiz (continued)

Cursor Basics

12. What is an explicit cursor?

13. Where do you define an explicit cursor?

14. Name the five steps for using an explicit cursor.

15. What is the syntax used to declare a cursor?

16. What does the `FOR UPDATE` clause do within a cursor definition?

17. What command opens an explicit cursor?

18. What command closes an explicit cursor?

19. Name five implicit actions that a cursor `FOR` loop provides.

20. Describe what the following cursor attributes do:
    - `cursor_name%ISOPEN`
    - `cursor_name%FOUND`
    - `cursor_name%NOTFOUND`
    - `cursor_name%ROWCOUNT`

Practice 1: PL/SQL Knowledge Quiz (continued)

Exceptions

21. An exception occurs in your PL/SQL block, which is enclosed in another PL/SQL block. What happens to this exception?

22. An exception handler is mandatory within a PL/SQL subprogram. (True/False)

23. What syntax do you use in the exception handler area of a subprogram?

24. How do you code for a NO_DATA_FOUND error?

25. Name three types of exceptions.

26. To associate an exception identifier with an Oracle error code, what pragma do you use and where?

27. How do you explicitly raise an exception?

28. What types of exceptions are implicitly raised?

29. What does the raise_application_error procedure do?

Practice 1: PL/SQL Knowledge Quiz (continued)

Dependencies

30. Which objects can a procedure or function directly reference?

31. What are the two statuses that a schema object can have and where are they recorded?

32. The Oracle server automatically recompiles invalid procedures when they are called from the same _____. To avoid compile problems with remote database calls, you can use the _____ model instead of the timestamp model.

33. What data dictionary contains information on direct dependencies?

34. What script do you run to create the views `deptree` and `ideptree`?

35. What does the `deptree_fill` procedure do and what are the arguments that you need to provide?

Oracle-Supplied Packages

36. What does the `dbms_output` package do?

37. How do you write "This procedure works." from within a PL/SQL program by using the `dbms_output`?

38. What does `dbms_sql` do and how does this compare with Native Dynamic SQL?

Design Considerations

Objectives

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

- **Identify guidelines for cursor design**
- **Use cursor variables**
- **Create subtypes based on existing types for an application**

Oracle University and En-Sof Informatica E Treinamento Ltda use only

Objectives

This lesson discusses several concepts that apply to the designing of PL/SQL program units. This lesson explains how to:

- Design and use cursor variables
- Describe the predefined data types
- Create subtypes based on existing data types for an application

Guidelines for Cursor Design

Fetch into a record when fetching from a cursor.

```
DECLARE
  CURSOR cur_cust IS
    SELECT customer_id, cust_last_name, cust_email
    FROM customers
    WHERE credit_limit = 1200;
  v_cust_record     cur_cust%ROWTYPE;
BEGIN
  OPEN cur_cust;
  LOOP
    FETCH cur_cust INTO v_cust_record;
...
```

Guidelines for Cursor Design

When fetching from a cursor, fetch into a record. This way you do not need to declare individual variables, and you reference only the values you want to use. Additionally, you can automatically use the structure of the SELECT column list.

Guidelines for Cursor Design

Create cursors with parameters.

```
CREATE OR REPLACE PROCEDURE cust_pack
 (p_crd_limit_in NUMBER, p_acct_mgr_in NUMBER)
IS
  v_credit_limit NUMBER := 1500;
   CURSOR cur_cust
          (p_crd_limit NUMBER, p_acct_mgr NUMBER)
     IS
     SELECT customer_id, cust_last_name, cust_email
     FROM customers
     WHERE credit_limit = p_crd_limit
     AND   account_mgr_id = p_acct_mgr;
   cust_record    cur_cust%ROWTYPE;
BEGIN
   OPEN cur_cust(p_crd_limit_in, p_acct_mgr_in);
...
   CLOSE cur_cust;
...
   OPEN cur_cust(v_credit_limit, 145);
...
END;
```

Guidelines for Cursor Design (continued)

Whenever you have a need to use a cursor in more than one place with different values for the WHERE clause, create parameters for your cursor. Parameters increase the flexibility and reusability of cursors, because you can pass different values to the WHERE clause when you open a cursor, rather than hard-code a value for the WHERE clause.

Additionally, parameters help you avoid scoping problems, because the result set for the cursor is not tied to a specific variable in a program. You can define a cursor at a higher level and use it in any subblock with variables defined in the local block.

Guidelines for Cursor Design

Reference implicit cursor attributes immediately after the SQL statement executes.

```
BEGIN
  UPDATE customers
    SET   credit_limit = p_credit_limit
    WHERE customer_id  = p_cust_id;
  get_avg_order(p_cust_id);  -- procedure call
  IF SQL%NOTFOUND THEN
   ...
```

Guidelines for Cursor Design (continued)

If you are using an implicit cursor and reference a SQL cursor attribute, make sure you reference it immediately after a SQL statement is executed. This is because SQL cursor attributes are set on the result of the most recently executed SQL statement. The SQL statement can be executed in another program. Referencing a SQL cursor attribute immediately after a SQL statement executes ensures that you are dealing with the result of the correct SQL statement.

In the example in the slide, you cannot rely on the value of SQL%NOTFOUND for the UPDATE statement, because it is likely to be overwritten by the value of another SQL statement in the get_avg_order procedure. To ensure accuracy, the cursor attribute function SQL%NOTFOUND needs to be called immediately after the DML statement:

```
DECLARE
  v_flag BOOLEAN;
BEGIN
  UPDATE customers
    SET   credit_limit = p_credit_limit
    WHERE customer_id  = p_cust_id;
    v_flag := SQL%NOTFOUND
  get_avg_order(p_cust_id);  -- procedure call
  IF v_flag THEN
   ...
```

Guidelines for Cursor Design

Simplify coding with cursor FOR loops.

```
CREATE OR REPLACE PROCEDURE cust_pack
 (p_crd_limit_in NUMBER, p_acct_mgr_in NUMBER)
IS
  v_credit_limit NUMBER := 1500;
   CURSOR cur_cust
           (p_crd_limit NUMBER, p_acct_mgr NUMBER)
     IS
     SELECT customer_id, cust_last_name, cust_email
     FROM customers
     WHERE credit_limit = p_crd_limit
     AND   account_mgr_id = p_acct_mgr;
   cust_record    cur_cust%ROWTYPE;
BEGIN
   FOR cust_record IN cur_cust
                   (p_crd_limit_in, p_acct_mgr_in)
  LOOP          -- implicit open and fetch
  ...
  END LOOP;     -- implicit close
  ...
END;
```

Guidelines for Cursor Design (continued)

Whenever possible, use cursor FOR loops that simplify coding. Cursor FOR loops reduce the volume of code you need to write to fetch data from a cursor and also reduce the chances of introducing loop errors in your code.
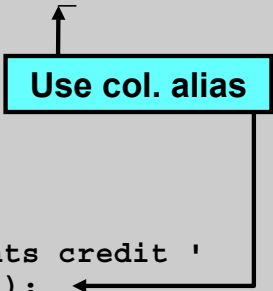
A cursor FOR loop automatically handles the open, fetch, and close operations, as well as, defines a record type that matches the cursor definition. After it processes the last row the cursor is closed automatically. If you do not use a CURSOR FOR loop, forgetting to close your cursor results in increased memory usage.

Guidelines for Cursor Design

- **Close a cursor when it is no longer needed.**
- **Use column aliases in cursors for calculated columns fetched into records declared with `%ROWTYPE`.**

```
CREATE OR REPLACE PROCEDURE cust_list
IS
 CURSOR  cur_cust IS
  SELECT customer_id, cust_last_name, credit_limit*1.1
  FROM    customers;
 cust_record cur_cust%ROWTYPE;
BEGIN
  OPEN cur_cust;
  LOOP
    FETCH cur_cust INTO cust_record;
    DBMS_OUTPUT.PUT_LINE('Customer ' ||
      cust_record.cust_last_name || ' wants credit '
      || cust_record.(credit_limit * 1.1));
    EXIT WHEN cur_cust%NOTFOUND;
  END LOOP;
...
```

Use col. alias

Guidelines for Cursor Design (continued)

- If you do not need a cursor any longer, close it explicitly. If your cursor is in a package, its scope is not limited to any particular PL/SQL block. The cursor remains open until you explicitly close it. An open cursor takes up memory space and continues to maintain row-level locks, if created with the `FOR UPDATE` clause, until a commit or rollback. Closing the cursor releases memory. Ending the transaction by committing or rolling back releases the locks. Along with a `FOR UPDATE` clause you can also use a `WHERE CURRENT OF` clause with the DML statements inside the `FOR` loop. This automatically performs a DML transaction for the current row in the cursor's result set, thereby improving performance. **Note:** It is a good programming practice to explicitly close your cursors. Leaving cursors open can generate an exception because the number of cursors allowed to remain open within a session is limited.
- Make sure that you use column aliases in your cursor for calculated columns that you fetch into a record declared with a `%ROWTYPE` declaration. You also need column aliases if you want to reference the calculated column in your program.
  The code in the slide does not compile successfully because it lacks a column alias for the calculation `credit_limit*1.1`. After you give it an alias, use the same alias later in the code to make a reference to the calculation.

Cursor Variables

Memory

1	Southlake, Texas	1400
2	San Francisco	1500
3	New Jersey	1600
4	Seattle, Washington	1700
5	Toronto	1800

REF CURSOR memory locator

Cursor Variables: Overview

Like a cursor, a cursor variable points to the current row in the result set of a multirow query. Cursor variables, however, are like C pointers: they hold the memory location of an item instead of the item itself. In this way, cursor variables differ from cursors the way constants differ from variables. A cursor is static, a cursor variable is dynamic. In PL/SQL, a cursor variable has a **REF CURSOR** data type, where REF stands for reference, and CURSOR stands for the class of the object.

Using Cursor Variables

To execute a multirow query, the Oracle server opens a work area called a "cursor" to store processing information. To access the information, you either explicitly name the work area, or you use a cursor variable that points to the work area. Whereas a cursor always refers to the same work area, a cursor variable can refer to different work areas. Therefore, cursors and cursor variables are not interoperable.

An explicit cursor is static and is associated with one SQL statement. A cursor variable can be associated with different statements at run time.

Primarily you use a cursor variable to pass a pointer to query result sets between PL/SQL stored subprograms and various clients such as a Developer Forms application. None of them owns the result set. They simply share a pointer to the query work area that stores the result set.

Oracle University and En-Sof Informatica E Treinamento Ltda   use only

Using a Cursor Variable

Define and declare the cursor variable.

①

Open the cursor variable.

②

Fetch rows from the result set.

③

Close the cursor variable.

④
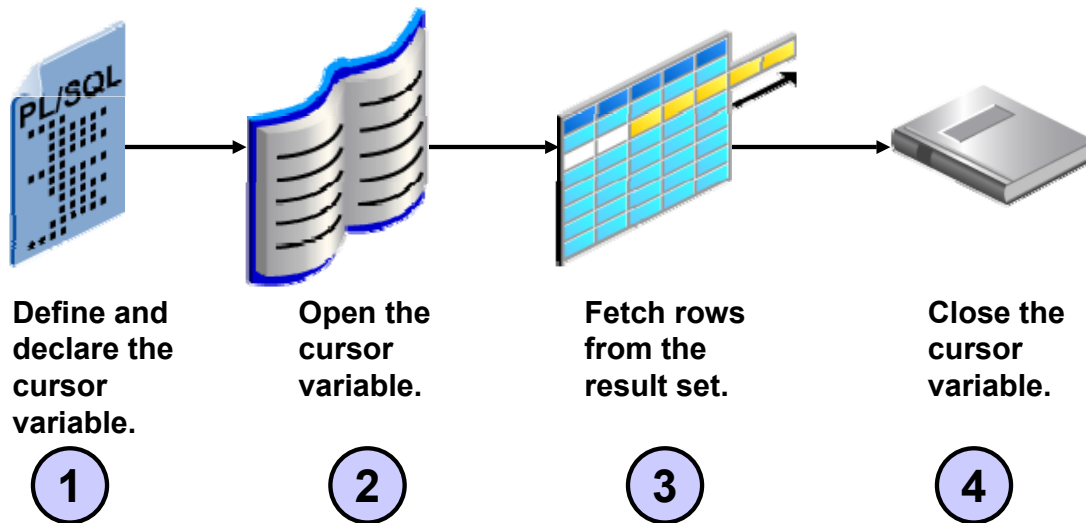
Working with Cursor Variables

There are four steps for handling a cursor variable. The next few sections contain detailed information about each step.

Strong Versus Weak Cursors

- **Strong cursor:**
  - **Is restrictive**
  - **Specifies a RETURN type**
  - **Associates with type-compatible queries only**
  - **Is less error prone**
- **Weak cursor:**
  - **Is nonrestrictive**
  - **Associates with any query**
  - **Is very flexible**

Strong Versus Weak Cursor Variables

REF CURSOR types can be strong (restrictive) or weak (nonrestrictive). A strong REF CURSOR type definition specifies a return type, a weak definition does not. PL/SQL enables you to associate a strong type with type-compatible queries only, whereas a weak type can be associated with any query. This makes strong REF CURSOR types less error prone, but weak REF CURSOR types more flexible.

In the following example, the first definition is strong, whereas the second is said to be weak:

```
DECLARE
   TYPE rt_cust IS REF CURSOR RETURN customers%ROWTYPE;
   TYPE rt_general_purpose IS REF CURSOR;
   ...
```

Step 1: Defining a REF CURSOR Type

Define a REF CURSOR type:

```
TYPE  ref_type_name IS REF CURSOR
   [RETURN return_type];
```

- *ref_type_name* **is a type specifier in subsequent declarations.**
- *return_type* **represents a record type.**
- *return_type* **indicates a strong cursor.**

```
DECLARE
  TYPE rt_cust IS REF CURSOR
    RETURN  customers%ROWTYPE;
  ...
```

Step 1: Defining a Cursor Variable

To create a cursor variable, you first need to define a REF CURSOR type and then declare a variable of that type.

Defining the REF CURSOR type:

```
TYPE  ref_type_name IS REF CURSOR  [RETURN return_type];
```

where: *ref_type_name*   a type specified in subsequent declarations

return_type   represents a row in a database table

The REF keyword indicates that the new type is to be a pointer to the defined type. The return_type is a record type indicating the types of the select list that are eventually returned by the cursor variable. The return type must be a record type.

Example

```
DECLARE
TYPE rt_cust IS REF CURSOR RETURN customers%ROWTYPE;
...
```

Step 1: Declaring a Cursor Variable

Declare a cursor variable of a cursor type:

```
CURSOR_VARIABLE_NAME      REF_TYPE_NAME
```

- *cursor_variable_name* **is the name of the cursor variable.**
- *ref_type_name* **is the name of a REF CURSOR type.**

```
DECLARE
  TYPE rt_cust IS REF CURSOR
    RETURN  customers%ROWTYPE;
  cv_cust rt_cust;
```

Declaring a Cursor Variable

After the cursor type is defined, declare a cursor variable of that type.

```
cursor_variable_name      ref_type_name;
```

where: *cursor_variable_name*   is the name of the cursor variable

ref_type_name   is the name of the REF CURSOR type

Cursor variables follow the same scoping and instantiation rules as all other PL/SQL variables.

In the following example, you declare the cursor variable cv_cust.

Step 1:

```
DECLARE
  TYPE ct_cust IS REF CURSOR RETURN customers%ROWTYPE;
  cv_cust rt_cust;
```

Step 1: Declaring a `REF CURSOR` Return Type

Options:

- **Use %`TYPE` and %`ROWTYPE`.**
- **Specify a user-defined record in the `RETURN` clause.**
- **Declare the cursor variable as the formal parameter of a stored procedure or function.**

Step 1: Declaring a `REF CURSOR` Return Type

The following are some more examples of cursor variable declarations:

- Use %`TYPE` and %`ROWTYPE` to provide the data type of a record variable:

```
DECLARE
 cust_rec customers%ROWTYPE; --a recd variable based on a row
 TYPE rt_cust IS REF CURSOR RETURN cust_rec%TYPE;
 cv_cust     rt_cust;  --cursor variable
```

- Specify a user-defined record in the `RETURN` clause:

```
DECLARE
  TYPE cust_rec_typ IS RECORD
   (custno     NUMBER(4),
    custname  VARCHAR2(10),
    credit    NUMBER(7,2));
  TYPE rt_cust IS REF  CURSOR RETURN cust_rec_typ;
  cv_cust  rt_cust;
```

- Declare a cursor variable as the formal parameter of a stored procedure or function:

```
DECLARE
  TYPE rt_cust IS REF  CURSOR RETURN customers%ROWTYPE;
  PROCEDURE use_cust_cur_var(cv_cust IN OUT  rt_cust)
  IS ...
```

Step 2: Opening a Cursor Variable

- **Associate a cursor variable with a multirow `SELECT` statement.**
- **Execute the query.**
- **Identify the result set:**

```
OPEN cursor_variable_name
    FOR select_statement
```

- *cursor_variable_name* **is the name of the cursor variable.**
- *select_statement* **is the SQL `SELECT` statement.**

Step 2: Opening a Cursor Variable

Other `OPEN-FOR` statements can open the same cursor variable for different queries. You do not need to close a cursor variable before reopening it. You must note that when you reopen a cursor variable for a different query, the previous query is lost.

In the following example, the packaged procedure declares a variable used to select one of several alternatives in an `IF THEN ELSE` statement. When called, the procedure opens the cursor variable for the chosen query.

```
CREATE OR REPLACE PACKAGE cust_data
IS
    TYPE rt_cust IS REF CURSOR RETURN customers%ROWTYPE;
    PROCEDURE open_cust_cur_var(cv_cust IN OUT rt_cust,
                                p_your_choice IN NUMBER);
END cust_data;
/
```

Step 2: Opening a Cursor Variable (continued)

```
CREATE OR REPLACE PACKAGE BODY cust_data
IS
    PROCEDURE open_cust_cur_var(cv_cust IN OUT rt_cust,
                                p_your_choice IN NUMBER)
    IS
    BEGIN
        IF p_your_choice = 1 THEN
           OPEN cv_cust FOR SELECT * FROM customers;
        ELSIF p_your_choice = 2 THEN
           OPEN cv_cust FOR SELECT * FROM customers
                                    WHERE credit_limit > 3000;
        ELSIF p_your_choice = 3 THEN
           ...
        END IF;
  END open_cust_cur_var;
END cust_data;
/
```

Step 3: Fetching from a Cursor Variable

- **Retrieve rows from the result set one at a time.**

```
FETCH cursor_variable_name
   INTO variable_name1
         [,variable_name2,. . .]
         | record_name;
```

- **The return type of the cursor variable must be compatible with the variables named in the INTO clause of the FETCH statement.**

Step 3: Fetching from a Cursor Variable

The FETCH statement retrieves rows from the result set one at a time. PL/SQL verifies that the return type of the cursor variable is compatible with the INTO clause of the FETCH statement. For each query column value returned, there must be a type-compatible variable in the INTO clause. Also, the number of query column values must equal the number of variables. In case of a mismatch in number or type, the error occurs at compile time for strongly typed cursor variables and at run time for weakly typed cursor variables.

Note: When you declare a cursor variable as the formal parameter of a subprogram that fetches from a cursor variable, you must specify the IN (or IN OUT) mode. If the subprogram also opens the cursor variable, you must specify the IN OUT mode.

Step 4: Closing a Cursor Variable

- **Disable a cursor variable.**
- **The result set is undefined.**

```
CLOSE cursor_variable_name ;
```

- **Accessing the cursor variable after it is closed raises the predefined exception INVALID_CURSOR.**

Step 4: Closing a Cursor Variable

The CLOSE statement disables a cursor variable. After that the result set is undefined. The syntax is:

```
CLOSE cursor_variable_name;
```

In the following example, the cursor is closed when the last row is processed.

```
...
  LOOP
    FETCH cv_cust INTO cust_rec;
    EXIT WHEN cv_cust%NOTFOUND;
     ...
  END LOOP;
  CLOSE cv_cust;
...
```

Passing Cursor Variables as Arguments

You can pass query result sets among PL/SQL stored subprograms and various clients.

Pointer to the result set

```
SQL> VARIABLE cv REFCURSOR
```

Access by a host variable on the client side

Passing Query Result Sets

Cursor variables are very useful for passing query result sets between PL/SQL stored subprograms and various clients. Neither PL/SQL nor any of its clients owns a result set; they simply share a pointer to the query work area that identifies the result set. For example, an Oracle Call Interface (OCI) client, or an Oracle Forms application, or the Oracle server can all refer to the same work area. This might be useful in Oracle Forms, for instance, when you want to populate a multiblock form.

Example

Using SQL*Plus, define a host variable with a data type of REFCURSOR to hold the query results generated from a REF CURSOR in a stored subprogram. Use the SQL*Plus PRINT command to view the host variable results. Optionally, you can set the SQL*Plus command SET AUTOPRINT ON to display the query results automatically.

```
SQL> VARIABLE cv REFCURSOR
```

Next, create a subprogram that uses a REF CURSOR to pass the cursor variable data back to the SQL*Plus environment.

Passing Cursor Variables as Arguments

```
SQL> EXECUTE cust_data.get_cust(112, :cv)

PL/SQL procedure successfully completed.


SQL> print cv

CUSTOMER_ID CUST_FIRST_NAME       CREDIT_LIMIT CUST_EMAIL
----------- -------------------- ------------ ----------------------------
        112 Guillaume                     200 Guillaume.Jackson@MOORHEN.COM
```

Passing Query Result Sets (continued)

```
CREATE OR REPLACE PACKAGE cust_data AS
TYPE typ_cust_rec IS RECORD
  (cust_id NUMBER(6), custname VARCHAR2(20),
   credit   NUMBER(9,2), cust_email VARCHAR2(30));
TYPE rt_cust IS REF CURSOR RETURN typ_cust_rec;
PROCEDURE get_cust
 (p_custid IN NUMBER, p_cv_cust IN OUT rt_cust);
END;
/
```

Passing Query Result Sets (continued)

```
CREATE OR REPLACE PACKAGE BODY cust_data AS
 PROCEDURE get_cust
   (p_custid IN NUMBER, p_cv_cust IN OUT rt_cust)
 IS
 BEGIN
   OPEN p_cv_cust FOR
SELECT customer_id, cust_first_name, credit_limit, cust_email
    FROM customers
    WHERE customer_id = p_custid;
-- CLOSE p_cv_cust
 END;
END;
/
```

Note that the CLOSE p_cv_cust statement is commented. This is done because if you close the REF cursor, it is not accessible from the host variable.

Rules for Cursor Variables

- **Cursor variables cannot be used with remote subprograms on another server.**
- **The query associated with a cursor variable in an `OPEN-FOR` statement should not be `FOR UPDATE`.**
- **You cannot use comparison operators to test cursor variables.**
- **Cursor variables cannot be assigned a null value.**
- **You cannot use `REF CURSOR` types in `CREATE TABLE` or `VIEW` statements.**
- **Cursors and cursor variables are not interoperable.**

Restrictions

- Remote subprograms on another server cannot accept the values of cursor variables. Therefore, you cannot use remote procedure calls (RPCs) to pass cursor variables from one server to another.
- If you pass a host cursor variable to PL/SQL, you cannot fetch from it on the server side unless you open it in the server on the same server call.
- The query associated with a cursor variable in an `OPEN-FOR` statement should not be `FOR UPDATE`.
- You cannot use comparison operators to test cursor variables for equality, inequality, or nullity.
- You cannot assign NULLs to a cursor variable.
- You cannot use `REF CURSOR` types to specify column types in a `CREATE TABLE` or `CREATE VIEW` statement. So, database columns cannot store the values of cursor variables.
- You cannot use a `REF CURSOR` type to specify the element type of a collection, which means that elements in an index-by table, nested table, or `VARRAY` cannot store the values of cursor variables.
- Cursors and cursor variables are not interoperable; that is, you cannot use one where the other is expected.

Comparing Cursor Variables
with Static Cursors

Cursor variables have the following benefits:

- **Are dynamic and ensure more flexibility**
- **Are not tied to a single SELECT statement**
- **Hold the value of a pointer**
- **Can reduce network traffic**
- **Give access to query work area after a block completes**

Comparing Cursor Variables with Static Cursors

Cursor variables are dynamic and provide wider flexibility. Unlike static cursors, cursor variables are not tied to a single SELECT statement. In applications where the SELECT statement may differ depending on different situations, cursor variables can be opened for the different SELECT statement. Because cursor variables hold the value of a pointer, they can be easily passed between programs, no matter where the programs exist.

Cursor variables can reduce network traffic by grouping OPEN FOR statements and sending them across the network only once. For example, the following PL/SQL block opens two cursor variables in a single round trip:

```
/* anonymous PL/SQL block in host environment */
BEGIN
    OPEN :cv_cust FOR SELECT * FROM customers;
    OPEN :cv_orders FOR SELECT * FROM orders;
END;
```

This may be useful in Oracle Forms, for instance, when you want to populate a multiblock form. When you pass host cursor variables to a PL/SQL block for opening, the query work areas to which they point remain accessible after the block completes. That enables your OCI or Pro*C program to use these work areas for ordinary cursor operations.

Predefined Data Types

Scalar Types		Composite Types
BINARY_DOUBLE **BINARY_FLOAT** **BINARY_INTEGER** **DEC** **DECIMAL** **DOUBLE_ PRECISION** **FLOAT** **INT** **INTEGER** **NATURAL** **NATURALN** **NUMBER** **NUMERIC** **PLS_INTEGER** **POSITIVE** **POSITIVEN** **REAL** **SINGTYPE** **SMALLINT**	**CHAR** **CHARACTER** **LONG** **LONG RAW** **NCHAR** **NVARCHAR2** **RAW** **ROWID** **STRING** **UROWID** **VARCHAR** **VARCHAR2** **BOOLEAN** **DATE**	**RECORD** **TABLE** **VARRAY** **Reference Types** **REF CURSOR** **REF object_type** **LOB Types** **BFILE** **BLOB** **CLOB** **NCLOB**
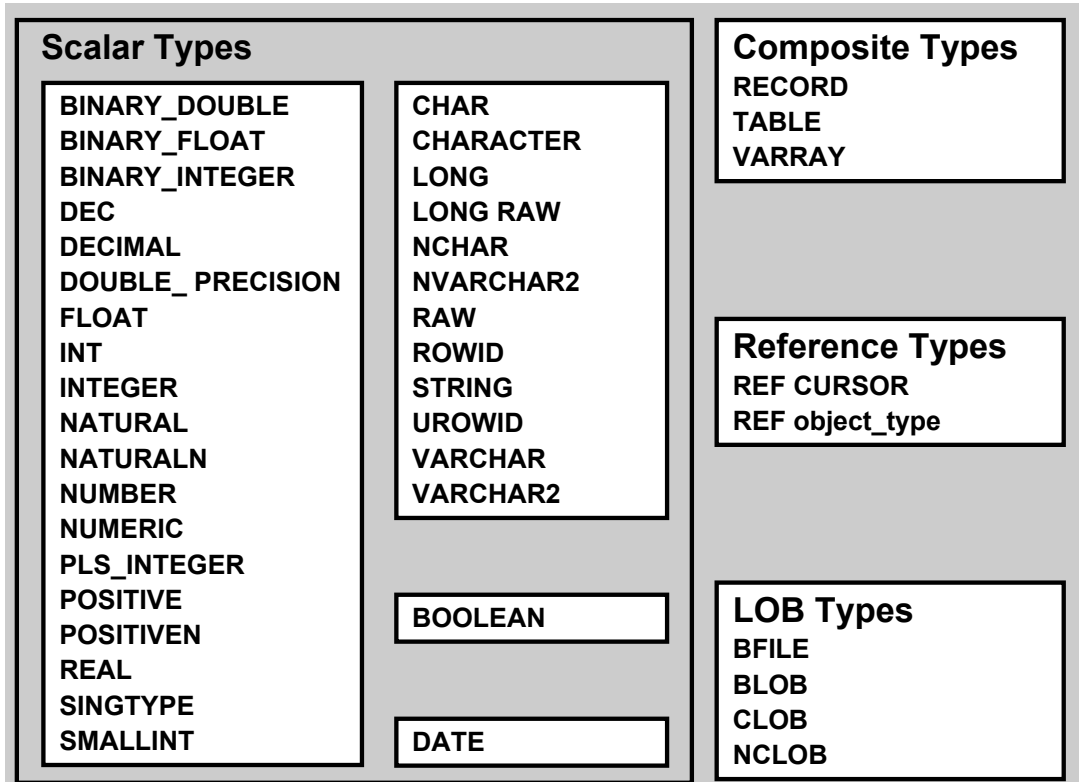
ORACLE

PL/SQL Data Types

Every constant, variable, and parameter has a data type, which specifies a storage format, constraints, and a valid range of values. PL/SQL provides a variety of predefined data types. For instance, you can choose from integer, floating point, character, Boolean, date, collection, reference, and LOB types. In addition, PL/SQL enables you to define your own subtypes.

Subtypes

A subtype is a subset of an existing data type that may place a constraint on its base type.
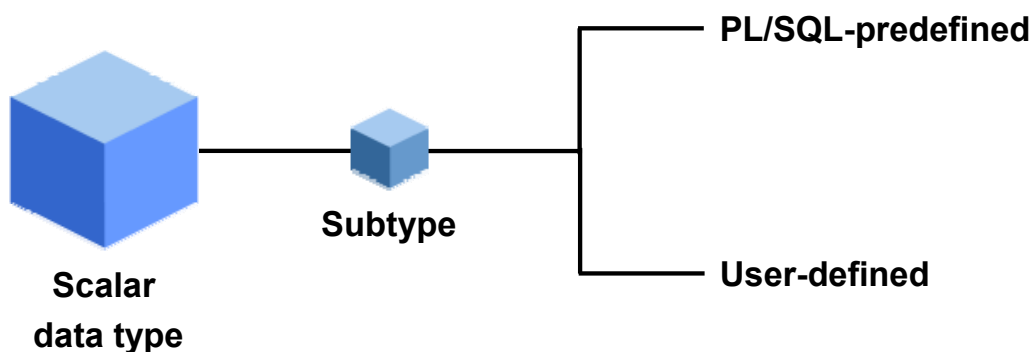
Scalar data type → Subtype → PL/SQL-predefined / User-defined

Definition of Subtypes

A subtype is a data type based on an existing data type. It does not define a new data type, instead it places a constraint on an existing data type. There are several predefined subsets specified in the standard package. DECIMAL and INTEGER are subtypes of NUMBER. CHARACTER is a subtype of CHAR.

Standard Subtypes

BINARY_INTEGER Subtypes	NUMBER Subtypes	VARCHAR2 Subtypes
NATURAL NATURALN POSITIVE POSITIVEN SIGNTYPE	DEC DECIMAL DOUBLE PRECISION FLOAT INTEGER INT NUMERIC REAL SMALLINT	STRING VARCHAR

Definition of Subtypes (continued)

With the NATURAL and POSITIVE subtypes, you can restrict an integer variable to non-negative and positive values, respectively. NATURALN and POSITIVEN prevent the assigning of nulls to an integer variable. You can use SIGNTYPE to restrict an integer variable to the values –1, 0, and 1, which is useful in programming tri-state logic.

A constrained subtype is a subset of the values normally specified by the data type on which the subtype is based. POSITIVE is a constrained subtype of BINARY_INTEGER.

An unconstrained subtype is not a subset of another data type; it is an alias to another data type. FLOAT is an unconstrained subtype of NUMBER.

Use the subtypes DEC, DECIMAL, and NUMERIC to declare fixed-point numbers with a maximum precision of 38 decimal digits.

Use the subtypes DOUBLE PRECISION and FLOAT to declare floating-point numbers with a maximum precision of 126 binary digits, which is roughly equivalent to 38 decimal digits. Or, use the subtype REAL to declare floating-point numbers with a maximum precision of 63 binary digits, which is roughly equivalent to 18 decimal digits.

Use the subtypes INTEGER, INT, and SMALLINT to declare integers with a maximum precision of 38 decimal digits.

You can create your own user-defined subtypes.

Note: You can use these subtypes for compatibility with ANSI/ISO and IBM types. Currently, VARCHAR is synonymous with VARCHAR2. However, in future releases of PL/SQL, to accommodate emerging SQL standards, VARCHAR may become a separate data type with different comparison semantics. It is a good idea to use VARCHAR2 rather than VARCHAR.

Benefits of Subtypes

Subtypes:

- **Increase reliability**
- **Provide compatibility with ANSI/ISO and IBM types**
- **Promote reusability**
- **Improve readability**
  - **Clarity**
  - **Code self-documents**

Benefits

If your applications require a subset of an existing data type, you can create your own subtypes. By using subtypes, you can increase the reliability and improve the readability by indicating the intended use of constants and variables. Subtypes can increase reliability by detecting the out-of-range values.

With the predefined subtypes, you have compatibility with other data types from other programming languages.

Declaring Subtypes

- **Subtypes are defined in the declarative section of any PL/SQL block.**

```
SUBTYPE subtype_name IS base_type [(constraint)]
[NOT NULL];
```

- *subtype_name* **is a type specifier used in subsequent declarations.**
- *base_type* **is any scalar or user-defined PL/SQL type.**

Declaring Subtypes

Subtypes are defined in the declarative section of a PL/SQL block, subprogram, or package.

Using the SUBTYPE keyword, you name the subtype and provide the name of the base type. The base type may be constrained starting in Oracle8*i*, but cannot be constrained in earlier releases.

You can use the %TYPE attribute on the base type to pick up a data type from a database column or from an existing variable data type. You can also use the %ROWTYPE attribute.

Examples

```
CREATE OR REPLACE PACKAGE mytypes
IS
  SUBTYPE  Counter IS INTEGER;  -- based on INTEGER type
  TYPE typ_TimeRec IS RECORD (minutes INTEGER, hours
  INTEGER);
  SUBTYPE Time IS typ_TimeRec;   -- based on RECORD type
  SUBTYPE ID_Num IS customers.customer_id%TYPE;
  CURSOR cur_cust IS SELECT * FROM customers;
  SUBTYPE CustFile IS cur_cust%ROWTYPE; -- based on cursor
END mytypes;
/
```

Using Subtypes

- **Define an identifier that uses the subtype in the declarative section.**

```
identifier_name   subtype_name
```

- **You can constrain a user-defined subtype when declaring variables of that type.**

```
identifier_name   subtype_name(size)
```

- **You can constrain a user-defined subtype when declaring the subtype.**

Using Subtypes

After the subtype is declared, you can assign an identifier for that subtype. Subtypes can increase reliability by detecting out-of-range values.

```
DECLARE
  v_rows     mytypes.Counter; --use package subtype dfn
  v_customers  mytypes.Counter;
  v_start_time mytypes.Time;
  SUBTYPE    Accumulator IS NUMBER;
  v_total      Accumulator(4,2);
SUBTYPE    Scale IS NUMBER(1,0);   -- constrained subtype
  v_x_axis     Scale;  -- magnitude range is -9 .. 9
BEGIN
  v_rows := 1;
  v_start_time.minutes := 15;
  v_start_time.hours   := 03;
 dbms_output.put_line('Start time is: '||
 v_start_time.hours|| ':' || v_start_time.minutes);
END;
/
```

Subtype Compatibility

An unconstrained subtype is interchangeable with its base type.

```
DECLARE
  SUBTYPE Accumulator IS NUMBER;
  v_amount   NUMBER(4,2);
  v_total    Accumulator;
BEGIN
  v_amount := 99.99;
  v_total  := 100.00;
  dbms_output.put_line('Amount is: ' || v_amount);
  dbms_output.put_line('Total is: ' || v_total);
  v_total := v_amount;
  dbms_output.put_line('This works too: ' ||
  v_total);
  --  v_amount := v_amount + 1; Will show value error
END;
/
```

Type Compatibility

An unconstrained subtype is interchangeable with its base type. Different subtypes are interchangeable if they have the same base type. Different subtypes are also interchangeable if their base types are in the same data type family.

```
DECLARE
    v_rows         mytypes.Counter;
    v_customers    mytypes.Counter;
    SUBTYPE   Accumulator IS NUMBER;
    v_total        Accumulator(6,2);
BEGIN
    SELECT COUNT(*) INTO v_customers FROM customers;
    SELECT COUNT(*) INTO v_rows FROM orders;
    v_total := v_customers + v_rows;
    DBMS_OUTPUT.PUT_LINE('Total rows from 2 tables: '||
    v_total);
EXCEPTION
    WHEN value_error THEN
    DBMS_OUTPUT.PUT_LINE('Error in data type.');
END;
/
```

Summary

In this lesson, you should have learned how to:

- **Use guidelines for cursor design**
- **Declare, define, and use cursor variables**
- **Use subtypes as data types**

Summary

- Use the guidelines for designing the cursors
- Take advantage of the features of cursor variables and pass pointers to result sets to different applications.
- You can use subtypes to organize and strongly type data types for an application.

Practice Overview

This practice covers the following topics:

- **Determining the output of a PL/SQL block**
- **Improving the performance of a PL/SQL block**
- **Implementing subtypes**
- **Using cursor variables**

ORACLE

Practice Overview

In this practice you will determine the output of a PL/SQL code snippet and modify the snippet to improve the performance. Next, you will implement subtypes and use cursor variables to pass values to and from a package.

Practice 2

Note: You will be using `oe/oe` as the `username/password` for the practice exercises. Files mentioned in the practice exercises are found in `/labs` folder. Additionally, solution scripts are provided for each question and are located in the `/soln` folder. Your instructor will provide you with the exactly location of these files.

1. Determine the output of the following code snippet.

```
SET SERVEROUTPUT ON
BEGIN
   UPDATE orders SET order_status = order_status;
   FOR v_rec IN ( SELECT order_id FROM orders )
   LOOP
      IF SQL%ISOPEN THEN
         DBMS_OUTPUT.PUT_LINE('TRUE - ' || SQL%ROWCOUNT);
      ELSE
         DBMS_OUTPUT.PUT_LINE('FALSE - ' || SQL%ROWCOUNT);
      END IF;
   END LOOP;
END;
/
```

2. Modify the following snippet of code to make better use of the `FOR UPDATE` clause and improve the performance of the program.

```
DECLARE
   CURSOR cur_update
     IS SELECT * FROM customers
     WHERE credit_limit < 5000 FOR UPDATE;
BEGIN
   FOR v_rec IN cur_update
   LOOP
      IF v_rec IS NOT NULL
      THEN
            UPDATE customers
            SET credit_limit = credit_limit + 200
            WHERE customer_id = v_rec.customer_id;
      END IF;
   END LOOP;
END;
/
```

Practice 2 (continued)

3. Create a package specification that defines subtypes, which can be used for the `warranty_period` field of the `product_information` table. Name this package `MY_TYPES`. The type needs to hold the month and year for a warranty period.

4. Create a package named `SHOW_DETAILS` that contains two subroutines. The first subroutine should show order details for the given `order_id`. The second subroutine should show customer details for the given `customer_id`, including the customer Id, first name, phone numbers, credit limit, and email address. Both the subroutines should use the cursor variable to return the necessary details.

Working with Collections

3

Objectives

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

- **Describe an object type**
- **Create an object type specification**
- **Implement the constructor method on objects**
- **Create collections**
  - **Nested table collections, varray collections**
  - **Associative arrays, string indexed collections**
- **Use collections methods**
- **Manipulate collections**
- **Distinguish between the different types of collections and when to use them**

ORACLE

Objectives

In this lesson, you are introduced to PL/SQL programming using collections, including user-defined object types and constructor methods.

Oracle object types are user-defined data types that make it possible to model complex real-world entities such as customers and purchase orders as unitary entities—objects—in the database.

A collection is an ordered group of elements, all of the same type (for example, phone numbers for each customer). Each element has a unique subscript that determines its position in the collection.

Collections work like the set, queue, stack, and hash table data structures found in most third-generation programming languages. Collections can store instances of an object type and can also be attributes of an object type. Collections can be passed as parameters. So, you can use them to move columns of data into and out of database tables or between client-side applications and stored subprograms. You can define collection types in a PL/SQL package, then use the same types across many applications.

Understanding the Components
of an Object Type

- **An object type is a user-defined composite data type.**
- **An object type encapsulates a data structure.**

Attributes:

```
street address
 postal code
     city
state_province
  country_id
```

- **Object types can be transient or persistent.**

Object Types

Object types are abstractions of the real-world entities used in application programs. They are analogous to Java and C++ classes. You can think of an object type as a template, and an object as a structure that matches the template. Object types can represent many different data structures. Object types are schema objects, subject to the same kinds of administrative control as other schema objects.

An object, such as a car, an order, or a person, has specific attributes and behaviors. You use an object type to maintain this perspective. An object type is a user-defined composite data type that encapsulates a data structure.

- The variables that make up the data structure are called attributes.
- The data structure formed by a set of attributes is public (visible to client programs).

Persistent Versus Transient Objects

For persistent objects, the associated object instances are stored in the database. Persistent object types are defined in the database with the CREATE SQL statement. Transient objects are defined programmatically with PL/SQL and differ from persistent objects the way they are declared, initialized, used, and deleted. When the program unit finishes execution, the transient object no longer exists, but the type exists in the database.

Transient objects are defined as an instance of a persistent object type; therefore, transient object attributes cannot be PL/SQL data types.

Creating an Object Type

- **Syntax**

```
CREATE [OR REPLACE] TYPE type_name
  AS OBJECT
    ( attribute1 datatype,
      attribute2 datatype,
      ...
    );
```

- **Example of a persistent object type**

```
CREATE TYPE cust_address_typ
  AS OBJECT
    ( street_address      VARCHAR2(40)
    , postal_code         VARCHAR2(10)
    , city                VARCHAR2(30)
    , state_province      VARCHAR2(10)
    , country_id          CHAR(2)
    );
/
```

street address
postal code
city
state_province
country_id

ORACLE

Creating an Object Type

Creating an object type is similar to creating a package specification. In the object type definition, you list the attributes and data types for the object that you are creating (similar to defining variables in a package specification).

Object Methods

Object types can include procedures and functions to manipulate the object attributes. The procedures and functions that characterize the behavior are called methods. These methods are named when you define the object type. Another component of defining an object is creating the object type specification. It is similar to a package specification. In the object type specification, you define the code for the methods. Methods are covered in detail in the *Oracle Database Application Developer's Guide - Object-Relational Features* manual.

Using an Object Type

You can use an object type as an abstract data type for a column in a table:

```
CREATE TABLE customers
    ( customer_id        NUMBER(6)     ...
    , cust_first_name    VARCHAR2(20) ...
    , cust_last_name     VARCHAR2(20) ...
    , cust_address       cust_address_typ
    ...
```

```
DESCRIBE customers

Name                              Null?    Type
----------------------------- -------- ----------
CUSTOMER_ID                   NOT NULL NUMBER(6)
CUST_FIRST_NAME               NOT NULL VARCHAR2(20)
CUST_LAST_NAME                NOT NULL VARCHAR2(20)
CUST_ADDRESS                           CUST_ADDRESS_TYP
...
```

Using an Object Type

You can use object types as a column's data type. For example, in the CUSTOMERS table, the column CUST_ADDRESS has the data type of CUST_ADDRESS_TYP. This makes the CUST_ADDRESS column a multicelled field where each component is an attribute of the CUST_ADDRESS_TYP.

You can also use object types as abstract data types for variables in a PL/SQL subroutine. In this example, a local variable is defined to hold a value of CUST_ADDRESS_TYP type. This variable is a transient object—it exists for the duration of the execution of the program.

```
DECLARE
  v_address cust_address_typ;
BEGIN
  SELECT cust_address
    INTO v_address
    FROM customers
    WHERE customer_id = 101;
  DBMS_OUTPUT.PUT_LINE (v_address.street_address);
END;
/

514 W Superior St
PL/SQL procedure successfully completed.
```

Using Constructor Methods

```
INSERT INTO CUSTOMERS (customer_id, cust_first_name,
                       cust_last_name, cust_address)
VALUES (1000, 'John', 'Smith',
 cust_address_typ ( '285 Derby Street'
               , '02465'
               , 'Boston'
               , 'MA'
               , 'US'));
```

street address
postal code
city
state_province
country_id
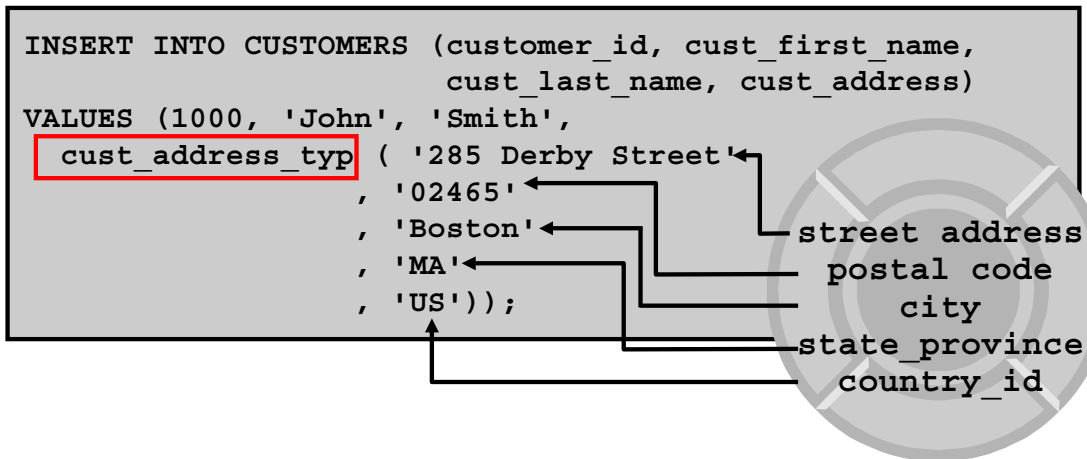
Using Constructor Methods

Every object type has a constructor method. A constructor is an implicitly defined function that is used to initialize an object. For arguments, it takes the values of the attributes for an object. PL/SQL never calls a constructor implicitly, so you must call it explicitly. You can make constructor calls wherever function calls are allowed. The constructor method has:

- The same name as the object type
- Formal parameters that exactly match the attributes of the object type (the number, order, and data types are the same)
- A return value of the given object type

In the example shown, the `cust_address_typ` constructor is used to initialize a row in the `cust_address` column of the `CUSTOMERS` table. The `cust_address_typ` constructor matches the `typ_cust_address` object and has five arguments that match the attributes for the `typ_cust_address` object.

Retrieving Data from Object Type Columns

```
SELECT customer_id, cust_first_name, cust_last_name, cust_address
FROM   customers
WHERE  customer_id = 1000;

CUSTOMER_ID CUST_FIRST_NAME      CUST_LAST_NAME
----------- -------------------- --------------------
CUST_ADDRESS(STREET_ADDRESS, POSTAL_CODE, CITY, STATE_PROVINCE, COUNTRY_ID)
--------------------------------------------------------------------------
       1000 John                 Smith
CUST_ADDRESS_TYP ('285 Derby Street', '02465', 'Boston', 'MA', 'US')
```

1

```
SELECT c.customer_id, c.cust_address.street_address,
       c.cust_address.city, c.cust_address.state_province,
       c.cust_address.postal_code, c.cust_address.country_id
FROM   customers c
WHERE  customer_id = 1000;

CUSTOMER_ID CUST_ADDRESS.STREET_ADDRESS
----------- ------------------------------------
CUST_ADDRESS.CITY               CUST_ADDRE CUST_ADDRE CU
------------------------------- ---------- ---------- --
       1000 285 Derby Street
Boston                          MA         02465      US
```

2

Retrieving Data from Object Type Columns

You retrieve information from object type columns by using a SELECT statement. You can view the results as a set of the constructor type (1), or in a flattened form (2).

The flattened form is useful when you access Oracle collection columns from relational tools and APIs, such as ODBC.

Understanding Collections

- **A collection is a group of elements, all of the same type.**
- **Collections work like arrays.**
- **Collections can store instances of an object type and, conversely, can be attributes of an object type.**
- **Types of collections in PL/SQL:**
  - **Nested tables**
  - **Varrays**
  - **Associative arrays**
  - **String indexed collections**
  - **INDEX BY pls_integer**

ORACLE

Collections

A collection is a group of elements, all of the same type. Each element has a unique subscript that determines its position in the collection. Collections work like the arrays found in most third-generation programming languages. They can store instances of an object type and, conversely, can be attributes of an object type. Collections can also be passed as parameters. You can use them to move columns of data into and out of database tables or between client-side applications and stored subprograms.

Object types are used not only to create object relational tables, but also to define collections.

You can use any of the three categories of collections:
- Nested tables can have any number of elements.
- A varray is an ordered collection of elements.
- Associative arrays (known as "index-by tables" in previous Oracle releases) are sets of key-value pairs, where each key is unique and is used to locate a corresponding value in the array. The key can be an integer or a string.

Note: Associative arrays indexed by integer are covered in the prerequisite courses: *Oracle Database 10g: Program with PL/SQL* and *Oracle Database 10g: Develop PL/SQL Program Units* and will not be emphasized in this course.

Oracle University and En-Sof Informatica E Treinamento Ltda use only

Describing the Collection Types
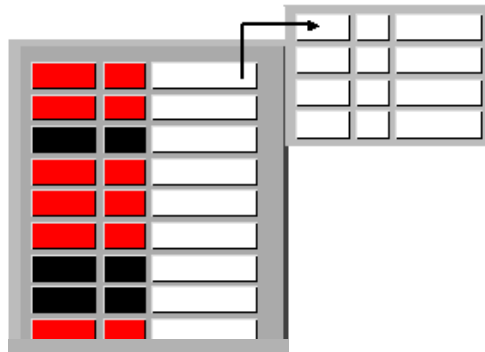
Nested table:

Varray:
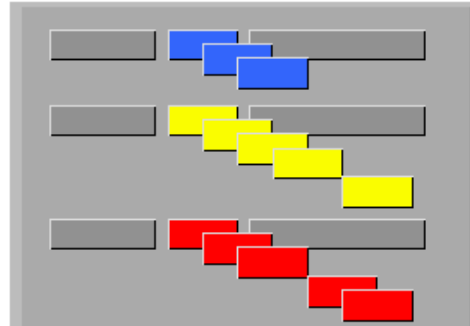
Associative array:

Index by
`PLS_INTEGER`
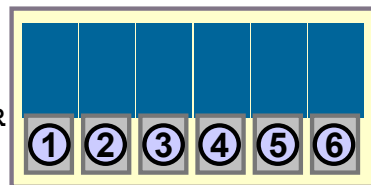
① ② ③ ④ ⑤ ⑥

ⓐ ⓕ ⓘ ⓞ ⓣ ⓦ

Index by
`VARCHAR2`
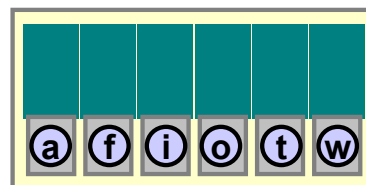
Collections (continued)

PL/SQL offers three collection types:

Nested Tables

A nested table holds a set of values. In other words, it is a table within a table. Nested tables are unbounded, meaning the size of the table can increase dynamically. Nested tables are available in both PL/SQL as well as the database. Within PL/SQL, nested tables are like one-dimensional arrays whose size can increase dynamically. Within the database, nested tables are column types that hold sets of values. The Oracle database stores the rows of a nested table in no particular order. When you retrieve a nested table from the database into a PL/SQL variable, the rows are given consecutive subscripts starting at 1. This gives you array-like access to individual rows. Nested tables are initially dense but they can become sparse through deletions and therefore have nonconsecutive subscripts.

Varrays

Variable-size arrays, or varrays, are also collections of homogeneous elements that hold a fixed number of elements (although you can change the number of elements at run time). They use sequential numbers as subscripts. You can define equivalent SQL types, allowing varrays to be stored in database tables. They can be stored and retrieved through SQL, but with less flexibility than nested tables. You can reference the individual elements for array operations, or manipulate the collection as a whole.

Collections (continued)

Varrays (continued)

Varrays are always bounded and never sparse. You can specify the maximum size of the varray in its type definition. Its index has a fixed lower bound of 1 and an extensible upper bound. A varray can contain a varying number of elements, from zero (when empty) to the maximum specified in its type definition.

To reference an element, you can use the standard subscripting syntax.

Associative Arrays

Associative arrays are sets of key-value pairs, where each key is unique and is used to locate a corresponding value in the array. The key can be either integer (PLS_INTEGER) or character (VARCHAR2) based.

When you assign a value using a key for the first time, it adds that key to the associative array. Subsequent assignments using the same key update the same entry. It is important to choose a key that is unique. For example, key values may come from the primary key of a database table, from a numeric hash function, or from concatenating strings to form a unique string value.

Because associative arrays are intended for storing temporary data rather than storing persistent data, you cannot use them with SQL statements such as INSERT and SELECT INTO. You can make them persistent for the life of a database session by declaring the type in a package and assigning the values in a package body.

Choosing a PL/SQL Collection Type

If you already have code or business logic that uses some other language, you can usually translate that language's array and set types directly to PL/SQL collection types.
- Arrays in other languages become varrays in PL/SQL.
- Sets and bags in other languages become nested tables in PL/SQL.
- Hash tables and other kinds of unordered lookup tables in other languages become associative arrays in PL/SQL.

If you are writing original code or designing the business logic from the start, consider the strengths of each collection type and decide which is appropriate.

Listing Characteristics for Collections

	PL/SQL Nested Tables	DB Nested Tables	PL/SQL Varrays	DB Varrays	PL/SQL Associative Arrays
Maximum size	No	No	Yes	Yes	Dynamic
Sparsity	Can be	No	Dense	Dense	Yes
Storage	N/A	Stored out of line	N/A	Stored in-line (if < 4,000 bytes)	N/A
Ordering	Does not retain ordering and subscripts	Does not retain ordering and subscripts	Retains ordering and subscripts	Retains ordering and subscripts	Retains ordering and subscripts

ORACLE

Choosing Between Nested Tables and Associative Arrays

- Use associative arrays when:
    - You need to collect information of unknown volume
    - You need flexible subscripts (negative, non-sequential, or string based)
    - You need to pass the collection to and from the database server (use associative arrays with the bulk constructs)
- Use nested tables when:
    - You need persistence
    - You need to pass the collection as a parameter

Choosing Between Nested Tables and Varrays

- Use varrays when:
    - The number of elements is known in advance
    - The elements are usually all accessed in sequence
- Use nested tables when:
    - The index values are not consecutive
    - There is no predefined upper bound for index values
    - You need to delete or update some elements, but not all the elements at once
    - You would usually create a separate lookup table, with multiple entries for each row of the main table, and access it through join queries

Using Collections Effectively

- **Varrays involve fewer disk accesses and are more efficient.**
- **Use nested tables for storing large amounts of data.**
- **Use varrays to preserve the order of elements in the collection column.**
- **If you do not have a requirement to delete elements in the middle of a collection, favor varrays.**
- **Varrays do not allow piecewise updates.**

ORACLE

Guidelines for Using Collections Effectively

- Because varray data is stored in-line (in the same tablespace), retrieving and storing varrays involves fewer disk accesses. Varrays are thus more efficient than nested tables.
- To store large amounts of persistent data in a column collection, use nested tables. This way the Oracle server can use a separate table to hold the collection data which can grow over time. For example, when a collection for a particular row could contain 1 to 1,000,000 elements, a nested table is simpler to use.
- If your data set is not very large and it is important to preserve the order of elements in a collection column, use varrays. For example, if you know that in each row the collection will not contain more than ten elements, you can use a varray with a limit of ten.
- If you do not want to deal with deletions in the middle of the data set, use varrays.
- If you expect to retrieve the entire collection simultaneously, use varrays.
- Varrays do not allow piecewise updates.

Note: If your application requires negative subscripts, you can use only associative arrays.

Creating Collection Types

Nested table in the database:

```
CREATE [OR REPLACE] TYPE type_name AS TABLE OF
Element_datatype [NOT NULL];
```

Nested table in PL/SQL:

```
TYPE type_name IS TABLE OF element_datatype
[NOT NULL];
```

Varray in the database:

```
CREATE [OR REPLACE] TYPE type_name AS VARRAY
(max_elements) OF element_datatype [NOT NULL];
```

Varray in PL/SQL:

```
TYPE type_name IS VARRAY  (max_elements) OF
element_datatype [NOT NULL];
```

Associative array in PL/SQL (string indexed):

```
TYPE type_name IS TABLE OF  (element_type)
INDEX BY VARCHAR2(size)
```

Creating Collection Types

To create a collection, you first define a collection type, and then declare collections of that type. The slide above shows the syntax for defining nested table and varray collection types in both the database (persistent) and in PL/SQL (transient), and defining a string indexed collection in PL/SQL.

Creating Collections in the Database

You can create a nested table or a varray data type in the database, which makes the data type available to use in places such as columns in database tables, variables in PL/SQL programs, and attributes of object types.

Before you can define a database table containing a nested table or varray, you must first create the data type for the collection in the database.

Use the syntax shown in the slide to create collection types in the database.

Creating Collections in PL/SQL

You can also create a nested table or a varray in PL/SQL. Use the syntax shown in the slide to create collection types in PL/SQL. You can create associative arrays in PL/SQL only.

Note: Collections can be nested. In Oracle9*i* and later, collections of collections are possible.

Declaring Collections: Nested Table

- **First, define an object type:**

```
CREATE TYPE typ_item AS OBJECT  --create object
  (prodid   NUMBER(5),
   price    NUMBER(7,2) )
/
CREATE TYPE typ_item_nst -- define nested table type
   AS TABLE OF typ_item
/
```
1
2

- **Second, declare a column of that collection type:**

```
CREATE TABLE pOrder (  -- create database table
     ordid    NUMBER(5),
     supplier NUMBER(5),
     requester        NUMBER(4),
     ordered DATE,
     items    typ_item_nst)
     NESTED TABLE items STORE AS item_stor_tab
/
```
3

Declaring Collections: Nested Table

To create a table based on a nested table, perform the following steps:

1. Create the typ_item type, which holds the information for a single line item.
2. Create the typ_item_nst type, which is created as a table of the typ_item type.
   Note: You must create the typ_item_nst nested table type based on the previously declared type because it is illegal to declare multiple data types in this nested table declaration.
3. Create the pOrder table and use the nested table type in a column declaration, which will include an arbitrary number of items based on the typ_item_nst type. Thus, each row of pOrder may contain a table of items.
   The NESTED TABLE STORE AS clause is required to indicate the name of the storage table in which the rows of all the values of the nested table reside. The storage table is created in the same schema and the same tablespace as the parent table.
   Note: The dictionary view USER_COLL_TYPES holds information about collections.

Understanding Nested Table Storage

Nested tables are stored out-of-line in storage tables.

pOrder nested table

Supplier	Requester	Ordered	Items
123	456	10-MAR-97	→
321	789	12-FEB-97	→

Storage table

NESTED_TABLE_ID	ProdID	Price
←	901	$ 45.95
←	879	$ 99.99
←	333	$ 0.22
←	112	$300.00

ORACLE

Nested Table Storage

The rows for all nested tables of a particular column are stored within the same segment. This segment is called the *storage table*.

A storage table is a system-generated segment in the database that holds instances of nested tables within a column. You specify the name for the storage table by using the NESTED TABLE STORE AS clause in the CREATE TABLE statement. The storage table inherits storage options from the outermost table.

To distinguish between nested table rows belonging to different parent table rows, a system-generated nested table identifier that is unique for each outer row enclosing a nested table is created.

Operations on storage tables are performed implicitly by the system. You should not access or manipulate the storage table, except implicitly through its containing objects.

Privileges of the column of the parent table are transferred to the nested table.

Declaring Collections: Varray

- **First, define a collection type:**

```
CREATE TYPE typ_Project AS OBJECT(   --create object
    project_no NUMBER(2),
    title      VARCHAR2(35),
    cost       NUMBER(7,2))
/
CREATE TYPE typ_ProjectList AS VARRAY (50) OF typ_Project
        -- define VARRAY type
/
```
1
2

- **Second, declare a collection of that type:**

```
CREATE TABLE department (  -- create database table
    dept_id  NUMBER(2),
    name     VARCHAR2(15),
    budget   NUMBER(11,2),
    projects typ_ProjectList)  -- declare varray as column
/
```
3

ORACLE

Example

The example above shows how to create a table based on a varray.
1. Create the `typ_project` type, which holds information for a project.
2. Create the `typ_ projectlist` type, which is created as a varray of the project type. The varray contains a maximum of 50 elements.
3. Create the `department` table and use the varray type in a column declaration. Each element of the varray will store a project object.

This example demonstrates how to create a varray of phone numbers, then use it in a `CUSTOMERS` table (The `OE` sample schema uses this definition.):

```
CREATE TYPE phone_list_typ
AS VARRAY(5) OF VARCHAR2(25);
/
CREATE TABLE customers
(customer_id NUMBER(6)
,cust_first_name VARCHAR2(50)
,cust_last_name VARCHAR2(50)
,cust_address cust_address_typ(100)
,phone_numbers phone_list_typ
...
);
```

Working with Collections in PL/SQL

- **You can declare collections as formal parameters of procedures and functions.**
- **You can specify a collection type in the `RETURN` clause of a function specification.**
- **Collections follow the usual scoping and instantiation rules.**

```
CREATE OR REPLACE PACKAGE manage_dept_proj AS
  TYPE typ_proj_details IS TABLE OF typ_Project;
  ...
  PROCEDURE allocate_proj
    (propose_proj IN typ_proj_details);
  FUNCTION top_project (n NUMBER)
    RETURN typ_proj_details;
  ...
```

ORACLE

Working with Collections

There are several points about collections that you must know when working with them:
- You can declare collections as the formal parameters of functions and procedures. That way, you can pass collections to stored subprograms and from one subprogram to another.
- A function's `RETURN` clause can be a collection type.
- Collections follow the usual scoping and instantiation rules. In a block or subprogram, collections are instantiated when you enter the block or subprogram and cease to exist when you exit. In a package, collections are instantiated when you first reference the package and cease to exist when you end the database session.

In the example in the slide, a nested table is used as the formal parameter of a packaged procedure, the data type of an `IN` parameter for the procedure `ALLOCATE_PROJ`, and the return data type of the `TOP_PROJECT` function.

Initializing Collections

Three ways to initialize:

- **Use a constructor.**
- **Fetch from the database.**
- **Assign another collection variable directly.**

```
DECLARE          --this example uses a constructor
  v_accounting_project typ_ProjectList;
BEGIN
  v_accounting_project :=
  typ_ProjectList
     (typ_Project (1, 'Dsgn New Expense Rpt', 3250),
      typ_Project (2, 'Outsource Payroll', 12350),
      typ_Project (3, 'Audit Accounts Payable',1425));
  INSERT INTO department
    VALUES(10, 'Accounting', 123, v_accounting_project);
...
END;
/
```

ORACLE

Initializing Collections

Until you initialize it, a collection is atomically null (that is, the collection itself is null, not its elements). To initialize a collection, you can use one of the following means:

- Use a constructor, which is a system-defined function with the same name as the collection type. A constructor allows the creation of an object from an object type. Invoking a constructor is a way to instantiate (create) an object. This function "constructs" collections from the elements passed to it. In the example shown above, you pass three elements to the `typ_ProjectList()` constructor, which returns a varray containing those elements.
- Read an entire collection from the database using a fetch.
- Assign another collection variable directly. You can copy the entire contents of one collection to another as long as both are built from the same data type.

Initializing Collections

```
DECLARE        -- this example uses a fetch from the database
  v_accounting_project typ_ProjectList;
BEGIN
  SELECT   projects
    INTO   v_accounting_project
    FROM   department
    WHERE  dept_id = 10;
...
END;
/
```
① ①

```
DECLARE -- this example assigns another collection
        -- variable directly
  v_accounting_project typ_ProjectList;
  v_backup_project       typ_ProjectList;
BEGIN
  SELECT   projects
    INTO   v_accounting_project
    FROM   department
    WHERE  dept_id = 10;
  v_backup_project := v_accounting_project;
END;
/
```
② ②

Initializing Collections (continued)

In the first example shown above, an entire collection from the database is fetched into the local PL/SQL collection variable.

In the second example shown above, the entire contents of one collection variable are assigned to another collection variable.

Referencing Collection Elements

Use the collection name and a subscript to reference a collection element:

- **Syntax:**

```
collection_name(subscript)
```

- **Example:**

```
v_accounting_project(1)
```

- **To reference a field in a collection:**

```
v_accounting_project(1).cost
```

Referencing Collection Elements

Every element reference includes a collection name and a subscript enclosed in parentheses. The subscript determines which element is processed. To reference an element, you can specify its subscript by using the following syntax:

```
collection_name(subscript)
```

In the preceding syntax, *subscript* is an expression that yields a positive integer. For nested tables, the integer must lie in the range 1 to 2147483647. For varrays, the integer must lie in the range 1 to maximum_size.

Using Collection Methods

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

```
collection_name.method_name [(parameters)]
```

Using Collection Methods

You can use collection methods from procedural statements but not from SQL statements.

Function or Procedure	Description
EXISTS	Returns TRUE if the nth element in a collection exists, otherwise, EXISTS(N) returns FALSE
COUNT	Returns the number of elements that a collection contains
LIMIT	For nested tables that have no maximum size, LIMIT returns NULL; for varrays, LIMIT returns the maximum number of elements that a varray can contain
FIRST and LAST	Returns the first and last (smallest and largest) index numbers in a collection, respectively
PRIOR and NEXT	PRIOR(n) returns the index number that precedes index n in a collection; NEXT(n) returns the index number that follows index n.
EXTEND	Appends one null element. EXTEND(n) appends n elements; EXTEND(n, i) appends n copies of the ith element
TRIM	Removes one element from the end; TRIM(n) removes n elements from the end of a collection
DELETE	Removes all elements from a nested or associative array table. DELETE(n) removes the nth element ; DELETE(m, n) removes a range. **Note:** Does not work on varrays.

Using Collection Methods

Traverse collections with methods:

```
DECLARE
  i INTEGER;
  v_accounting_project typ_ProjectList;
BEGIN
  v_accounting_project := typ_ProjectList(
    typ_Project (1,'Dsgn New Expense Rpt', 3250),
    typ_Project (2, 'Outsource Payroll', 12350),
    typ_Project (3, 'Audit Accounts Payable',1425));
  i := v_accounting_project.FIRST ;
  WHILE i IS NOT NULL LOOP
    IF v_accounting_project(i).cost > 10000 then
      DBMS_OUTPUT.PUT_LINE('Project too expensive: '
                       || v_accounting_project(i).title);
    END IF;
    i := v_accounting_project.NEXT (i);
  END LOOP;
END;
/
```

ORACLE

Traversing Collections

In the example shown, the FIRST method finds the smallest index number, the NEXT method traverses the collection starting at the first index. The output from this block of code shown above is:

```
Project too expensive: Outsource Payroll
```

You can use the PRIOR and NEXT methods to traverse collections indexed by any series of subscripts. In the example shown, the NEXT method is used to traverse a varray.

PRIOR(n) returns the index number that precedes index n in a collection. NEXT(n) returns the index number that succeeds index n. If n has no predecessor, PRIOR(n) returns NULL. Likewise, if n has no successor, NEXT(n) returns NULL. PRIOR is the inverse of NEXT.

PRIOR and NEXT do not wrap from one end of a collection to the other.

When traversing elements, PRIOR and NEXT ignore deleted elements.

Using Collection Methods

```
DECLARE
  v_my_projects  typ_ProjectList;
  v_array_count  INTEGER;
  v_last_element INTEGER;
BEGIN
  SELECT projects INTO v_my_projects FROM department
    WHERE dept_id = 10;
  v_array_count := v_my_projects.COUNT ;
  dbms_output.put_line('The # of elements is: ' ||
                        v_array_count);
  v_my_projects.EXTEND ;    --make room for new project
  v_last_element := v_my_projects.LAST ;
  dbms_output.put_line('The last element is: ' ||
                        v_last_element);
  IF v_my_projects.EXISTS(5) THEN
    dbms_output.put_line('Element 5 exists!');
  ELSE
    dbms_output.put_line('Element 5 does not exist.');
  END IF;
END;
/
```

ORACLE

Example

The block of code shown uses the COUNT, EXTEND, LAST, and EXISTS methods on the
my_projects varray. The COUNT method reports that the projects collection holds three
projects for department 10. The EXTEND method creates a fourth empty project. Using the LAST
method reports that four projects exist. When testing for the existence of a fifth project, the
program reports that it does not exist. The output from this block of code is as follows:

```
The # of elements is: 3
The last element is: 4
Element 5 does not exist.
PL/SQL procedure successfully completed.
```

Manipulating Individual Elements

```
CREATE OR REPLACE PROCEDURE add_project (
  p_deptno      IN NUMBER,
  p_new_project IN typ_Project,
  p_position    IN NUMBER )
IS
   v_my_projects typ_ProjectList;
BEGIN
  SELECT projects INTO v_my_projects FROM department
    WHERE dept_id = p_deptno FOR UPDATE OF projects;
  v_my_projects.EXTEND;    --make room for new project
  /* Move varray elements forward */
  FOR i IN REVERSE p_position..v_my_projects.LAST - 1 LOOP
    v_my_projects(i + 1) := v_my_projects(i);
  END LOOP;
  v_my_projects(p_position) := p_new_project; -- add new
                                              -- project
  UPDATE department SET projects = v_my_projects
    WHERE dept_id = p_deptno;
END add_project;
/
```

Manipulating Individual Elements

You must use PL/SQL procedural statements to reference the individual elements of a varray in an INSERT, UPDATE, or DELETE statement. In the example shown in the slide, the stored procedure inserts a new project into a department's project at a given position.

To execute the procedure, pass the department number to which you want to add a project, the project information, and the position where the project information is to be inserted.

```
        EXECUTE add_project(10, -
                  typ_Project(4, 'Information Technology', 789), 4)
        SELECT * FROM department;
          DEPT_ID NAME                    BUDGET
        ---------- --------------- ----------
        PROJECTS(PROJECT_NO, TITLE, COST)
        -------------------------------------------------------------
              10 Accounting              123
        PROJECTLIST(PROJECT(1, 'Dsgn New Expense Rpt', 3250),
        PROJECT(2, 'Outsource Payroll', 12350),
        PROJECT(3, 'Audit Accounts Payable', 1425),
        PROJECT(4, 'Information Technology', 789))
```

Avoiding Collection Exceptions

Common exceptions with collections:

- `COLLECTION_IS_NULL`
- `NO_DATA_FOUND`
- `SUBSCRIPT_BEYOND_COUNT`
- `SUBSCRIPT_OUTSIDE_LIMIT`
- `VALUE_ERROR`

Avoiding Collection Exceptions

In most cases, if you reference a nonexistent collection element, PL/SQL raises a predefined exception.

Exception	Raised when:
`COLLECTION_IS_NULL`	You try to operate on an atomically null collection
`NO_DATA_FOUND`	A subscript designates an element that was deleted
`SUBSCRIPT_BEYOND_COUNT`	A subscript exceeds the number of elements in a collection
`SUBSCRIPT_OUTSIDE_LIMIT`	A subscript is outside the legal range
`VALUE_ERROR`	A subscript is null or not convertible to an integer

Avoiding Collection Exceptions

Common exceptions with collections:

```
DECLARE
  TYPE NumList IS TABLE OF NUMBER;
  nums NumList;            -- atomically null
BEGIN
  /* Assume execution continues despite the raised
exceptions. */
  nums(1) := 1;           -- raises COLLECTION_IS_NULL
  nums := NumList(1,2); -- initialize table
  nums(NULL) := 3         -- raises VALUE_ERROR
  nums(0) := 3;           -- raises SUBSCRIPT_OUTSIDE_LIMIT
  nums(3) := 3;           -- raises SUBSCRIPT_BEYOND_COUNT
  nums.DELETE(1);         -- delete element 1
  IF nums(1) = 1 THEN   -- raises NO_DATA_FOUND
...
```

Example

In the first case, the nested table is atomically null. In the second case, the subscript is null. In the third case, the subscript is outside the legal range. In the fourth case, the subscript exceeds the number of elements in the table. In the fifth case, the subscript designates a deleted element.

Working with Collections in SQL

Querying collections:

```
SELECT * FROM department;

  DEPT_ID NAME                  BUDGET
--------- --------------- ----------
PROJECTS(PROJECT_NO, TITLE, COST)
-----------------------------------------------------------
       10 Accounting             123
PROJECTLIST(PROJECT(1, 'Dsgn New Expense Rpt', 3250), ...
oll', 12350), PROJECT(3, 'Audit Accounts Payable', ...
```

- **Querying a collection column in the `SELECT` list nests the elements of the collection in the result row with which the collection is associated.**
- **To unnest results from collection queries, use the `TABLE` expression in the `FROM` clause.**

ORACLE

Querying Collections

You can use two general ways to query a table that contains a column or attribute of a collection type. One way returns the collections nested in the result rows that contain them. By including the collection column in the `SELECT` list, the output shows as a row associated with the other row output in the `SELECT` list.

Another method to display the output is to unnest the collection such that each collection element appears on a row by itself. You can use the `TABLE` expression in the `FROM` clause to unnest a collection.

Working with Collections in SQL

`TABLE` expression:

```
SELECT d1.dept_id, d1.budget, d2.*
FROM   department d1, TABLE(d1.projects) d2;


 DEPT_ID  BUDGET PROJECT_NO TITLE                               COST
 ------- ------- ---------- --------------------------- ----------
      10     123          1 Dsgn New Expense Rpt             3250
      10     123          2 Outsource Payroll               12350
      10     123          3 Audit Accounts Payable           1425
      10     123          4 Information Technology            789
```

- **Enables you to query a collection in the `FROM` clause like a table**
- **Can be used to query any collection value expression, including transient values such as variables and parameters**

ORACLE

Querying Collections with the `TABLE` Expression

To view collections in a conventional format, you must unnest, or flatten, the collection attribute of a row into one or more relational rows. You can do this by using a TABLE expression with the collection. A TABLE expression enables you to query a collection in the FROM clause like a table. In effect, you join the nested table with the row that contains the nested table without writing a JOIN statement.

The collection column in the TABLE expression uses a table alias to identify the containing table.

You can use a subquery with the TABLE expression:

```
SELECT *
  FROM TABLE(SELECT d.projects
                FROM department d
                WHERE d.dept_id = 10);
```

You can use a TABLE expression in the FROM clause of a SELECT statement embedded in a CURSOR expression:

```
SELECT d.dept_id, CURSOR(SELECT * FROM TABLE(d.projects))
FROM   department d;
```

Working with Collections in SQL

- **The Oracle database supports the following DML operations on nested table columns:**
  - **Inserts and updates that provide a new value for the entire collection**
  - **Piecewise updates**
  - **Inserting new elements into the collection**
  - **Deleting elements from the collection**
  - **Updating elements of the collection**
- **The Oracle database does not support piecewise updates on varray columns.**
  - **Varray columns can be inserted into or updated as atomic units.**

DML Operations on Nested Table Columns

You can perform DML operations on nested table columns by providing inserts and updates that supply a new value for the entire collection. Previously, the `pOrder` table was defined with the `ITEMS` column as a nested table.

```
DESCRIBE pOrder
Name                              Null?    Type
------------------------- -------- -------------
ORDID                                     NUMBER(5)
SUPPLIER                                  NUMBER(5)
REQUESTER                                 NUMBER(4)
ORDERED                                   DATE
ITEMS                                     ITEM_NST_TYP
```

This example inserts rows by providing a new value for the entire collection:

```
INSERT INTO pOrder
  VALUES (500, 50, 5000, sysdate,
    typ_item_nst(typ_item (55, 555)));
1 row created.

INSERT INTO pOrder
  VALUES (800, 80, 8000, sysdate,
    typ_item_nst (typ_item (88, 888)));
1 row created.
```

Working with Collections in SQL

Piecewise DML on nested table columns:

- **INSERT**

```
INSERT INTO TABLE
   (SELECT p.items FROM pOrder p WHERE p.ordid = 500)
VALUES (44, 444);
```

- **UPDATE**

```
UPDATE TABLE
   (SELECT p.items FROM  pOrder p
    WHERE  p.ordid = 800) i
SET     VALUE(i) = typ_item(99, 999)
WHERE   i.prodid = 88;
```

- **DELETE**

```
DELETE FROM TABLE
  (SELECT p.items FROM pOrder p WHERE p.ordid = 500) i
WHERE i.prodid = 55;
```

ORACLE

DML on Nested Table Columns

For piecewise updates of nested table columns, the DML statement identifies the nested table value to be operated on by using the TABLE expression.

Using Set Operations on Collections

Oracle Database 10g supports the following multiset comparison operations on nested tables:

Set Operation	Description
Equal, Not equal Comparisons	The (=) and (<>) conditions return a Boolean value indicating whether the input nested tables are identical or not.
IN Comparisons	The IN condition checks whether a nested table is in a list of nested tables.
Subset of Multiset Comparison	The SUBMULTISET [OF] condition checks whether a nested table is a subset of another nested table.
Member of a Nested Table Comparison	The MEMBER [OF] or NOT MEMBER [OF] condition tests whether an element is a member of a nested table.
Empty Comparison	The IS [NOT] EMPTY condition checks whether a given nested table is empty or not empty, regardless of whether any of the elements are NULL.
Set Comparison	The IS [NOT] A SET condition checks whether a given nested table is composed of unique elements.

ORACLE

Comparisons of Collections

Starting with the Oracle Database 10g release, you can use comparison operators and ANSI SQL multiset operations on nested tables.

The conditions listed in this section allow comparisons of nested tables. There is no mechanism for comparing varrays.

In addition to the conditions listed in the slide, you can also use the multiset operations with nested tables:
- The CARDINALITY function returns the number of elements in a varray or nested table.
- The COLLECT function is an aggregate function which would create a multiset from a set of elements.
- The MULTISET EXCEPT operator inputs two nested tables and returns a nested table whose elements are in the first nested table but not in the second nested table.
- The MULTISET INTERSECT operator returns a nested table whose values are common in the two input nested tables.
- The MULTISET UNION operator returns a nested table whose values are those of the two input nested tables.

This list is not complete. For detailed information, refer to "Support for Collection Data types" in *Oracle Database Application Developer's Guide - Object-Relational Features.*

Using Set Operations on Collections

```
CREATE OR REPLACE TYPE billdate IS TABLE OF DATE;
/
ALTER TABLE pOrder ADD
  (notice billdate , payments billdate)
    NESTED TABLE notice   STORE AS notice_store_tab
    NESTED TABLE payments STORE AS payments_store_tab;
/
UPDATE pOrder
  SET notice   = billdate('15-JAN-02', '15-FEB-02', '15-MAR-02'),
      payments = billdate('15-FEB-02', '15-MAR-02', '15-JAN-02')
  WHERE ordid = 500;
```

```
SELECT   ordid
FROM     pOrder
WHERE    notice = payments;


 ORDID
------
   500
```

Equality and Non-Equality Predicates (= and <>)

The equal (=) and not equal (<>) operators can be used to compare nested tables. A Boolean result is returned from the comparison. Nested tables are considered equal if they have the same named type, the same cardinality, and their elements are equal. To make nested table comparisons, the element type needs to be comparable.

In this example, the pOrder table is altered. Two columns are added. Both columns are nested tables that hold the DATE data type. Dates are entered into the two columns for a specific order number. These two collections are compared with the equality predicate.

Using Set Operations on Collections

```
CREATE OR REPLACE TYPE typ_billdate IS TABLE OF DATE;
/
DECLARE
  b1       BOOLEAN;
  notice   typ_billdate;
  payments typ_billdate;
BEGIN
  notice   := typ_billdate('15-JAN-02','15-FEB-02','15-MAR-02');
  payments := typ_billdate('15-FEB-02','15-MAR-02','15-JAN-02');
  b1       := notice = payments;
  IF b1 THEN
    dbms_output.put_line('They are equal.');
  ELSE
    dbms_output.put_line('They are NOT equal.');
  END IF;
END;
/

Type created.
They are equal.
PL/SQL procedure successfully completed.
```

Set Operations in PL/SQL

In this example, two nested tables are defined in a PL/SQL block. Both are nested tables of the DATE data type. Date values are entered into the nested tables and the set of values in one nested table is compared using the equality operator with the set of values in the other nested table.

Using Multiset Operations on Collections

You can use the ANSI multiset comparison operations on nested table collections:

- **MULTISET EXCEPT**

- **MULTISET INTERSECT**

- **MULTISET UNION**

```
UPDATE pOrder
  SET notice   = billdate('31-JAN-02', '28-FEB-02', '31-MAR-02'),
      payments = billdate('28-FEB-02', '31-MAR-02')
    WHERE ordid = 500;
```

```
SELECT notice MULTISET INTERSECT payments
FROM pOrder ;

NOTICEMULTISETINTERSECTPAYMENTS
----------------------------------------------------------------
BILLDATE('28-FEB-02', '31-MAR-02')
```

Multiset Operations

In this example, the MULTISET INTERSECT operator finds the values that are common in the two input tables, NOTICE and PAYMENTS, and returns a nested table with the common results.

By using the CAST operator, you can convert collection-typed values of one type into another collection type. You can cast a named collection (such as a varray or a nested table) into a type-compatible named collection. For more information about the CAST operator with the MULTISET operand, see the topic "CAST" in *Oracle Database SQL Reference 10g Release 1*.

Using String Indexed Associative Arrays

Associative arrays:

- **Indexed by strings can improve performance**
- **Are pure memory structures that are much faster than schema-level tables**
- **Provide significant additional flexibility**
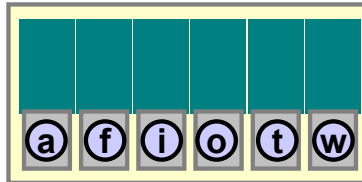
When to Use String Indexed Arrays

Starting with Oracle9*i* Database *Release 2*, you can use `INDEX BY VARCHAR2` tables (also known as string indexed arrays). These tables are optimized for efficiency by implicitly using the B*-tree organization of the values.

The `INDEX BY VARCHAR2` table is optimized for efficiency of lookup on a non-numeric key, where the notion of sparseness is not really applicable. In contrast, `INDEX BY PLS_INTEGER` tables are optimized for compactness of storage on the assumption that the data is dense.

Using String Indexed Associative Arrays

```
CREATE OR REPLACE PROCEDURE report_credit
  (p_last_name    customers.cust_last_name%TYPE,
   p_credit_limit customers.credit_limit%TYPE)
IS
  TYPE  typ_name IS TABLE OF customers%ROWTYPE
    INDEX BY customers.cust_email%TYPE;
  v_by_cust_email   typ_name;
  i VARCHAR2(30);

  PROCEDURE load_arrays IS
  BEGIN
    FOR rec IN  (SELECT * FROM customers WHERE cust_email IS NOT NULL)
      LOOP
        -- Load up the array in single pass to database table.
         v_by_cust_email (rec.cust_email) := rec;
      END LOOP;
  END;

BEGIN
  ...
```

Using String Indexed Arrays

If you need to do heavy processing of customer information in your program that requires going back and forth over the set of selected customers, you can use string indexed arrays to store, process, and retrieve the required information.

This can also be done in SQL, but probably in a less efficient implementation. If you need to do multiple passes over a significant set of static data, you can instead move it from the database into a set of collections. Accessing collection-based data is much faster than going through the SQL engine.

After you have transferred the data from the database to the collections, you can use string- and integer-based indexing on those collections to, in essence, mimic the primary key and unique indexes on the table.

In the REPORT_CREDIT procedure shown, you or a customer may need to determine whether a customer has adequate credit. The string indexed collection is loaded with the customer information in the LOAD_ARRAYS procedure. In the main body of the program, the collection is traversed to find the credit information. The e-mail name is reported in case there is more than one customer with the same last name.

Using String Indexed Associative Arrays

```
...
BEGIN
  load_arrays;
  i:= v_by_cust_email.FIRST;
  dbms_output.put_line ('For credit amount of: ' || p_credit_limit);
  WHILE i IS NOT NULL LOOP
    IF v_by_cust_email(i).cust_last_name = p_last_name
    AND v_by_cust_email(i).credit_limit > p_credit_limit
      THEN dbms_output.put_line ( 'Customer '||
        v_by_cust_email(i).cust_last_name || ': ' ||
        v_by_cust_email(i).cust_email ||  has credit limit of: ' ||
        v_by_cust_email(i).credit_limit);
    END IF;
    i := v_by_cust_email.NEXT(i);
  END LOOP;
END report_credit;
/
```

```
EXECUTE report_credit('Walken', 1200)
For credit amount of: 1200
Customer Walken: Emmet.Walken@LIMPKIN.COM has credit limit of: 3600
Customer Walken: Prem.Walken@BRANT.COM has credit limit of: 3700

PL/SQL procedure successfully completed.
```

Oracle University and En-Sof Informatica E Treinamento Ltda use only

Using String Indexed Arrays (continued)

In this example, the string indexed collection is traversed using the NEXT method.

A more efficient use of the string indexed collection is to index the collection with the customer e-mail. Then you can immediately access the information based on the customer e-mail key. You would need to pass the e-mail name instead of the customer last name.

Using String Indexed Arrays (continued)

Here is the modified code:

```
CREATE OR REPLACE PROCEDURE report_credit
  (p_email    customers.cust_last_name%TYPE,
   p_credit_limit customers.credit_limit%TYPE)
IS
  TYPE  typ_name IS TABLE OF customers%ROWTYPE
    INDEX BY customers.cust_email%TYPE;
  v_by_cust_email   typ_name;
  i VARCHAR2(30);

  PROCEDURE load_arrays IS
  BEGIN
    FOR rec IN  (SELECT * FROM customers
                  WHERE cust_email IS NOT NULL) LOOP
       v_by_cust_email (rec.cust_email) := rec;
    END LOOP;
  END;

BEGIN
  load_arrays;
  dbms_output.put_line
    ('For credit amount of: ' || p_credit_limit);
  IF v_by_cust_email(p_email).credit_limit > p_credit_limit
       THEN dbms_output.put_line ( 'Customer '||
         v_by_cust_email(p_email).cust_last_name ||
         ': ' || v_by_cust_email(p_email).cust_email ||
         ' has credit limit of: ' ||
         v_by_cust_email(p_email).credit_limit);
  END IF;
END report_credit;
/

EXECUTE report_credit('Prem.Walken@BRANT.COM', 100)
For credit amount of: 100
Customer Walken: Prem.Walken@BRANT.COM has credit limit of:
3700

PL/SQL procedure successfully completed.
```

Summary

In this lesson, you should have learned how to:
- **Identify types of collections**
  - **Nested tables**
  - **Varrays**
  - **Associative arrays**
- **Define nested tables and varrays in the database**
- **Define nested tables, varrays, and associative arrays in PL/SQL**
  - **Access collection elements**
  - **Use collection methods in PL/SQL**
  - **Identify raised exceptions with collections**
  - **Decide which collection type is appropriate for each scenario**

ORACLE

Summary

Collections are a grouping of elements, all of the same type. The types of collections are nested tables, varrays, and associative arrays. You can define nested tables and in the database. Nested tables, varrays, and associative arrays can be used in a PL/SQL program.

When using collections in PL/SQL programs, you can access collection elements, use predefined collection methods, and use exceptions that are commonly encountered with collections.

There are guidelines for using collections effectively and to determine which collection type is appropriate under specific circumstances.

Practice Overview

This practice covers the following topic:

- **Analyzing collections**
- **Using collections**

Practice Overview

In this practice, you will analyze collections for common errors, then you will create a collection and write a PL/SQL package to manipulate the collection.

For detailed instructions on performing this practice, see Appendix A, "Practice Solutions."

Practice 3

Collection Analysis

1. Examine the following definitions. Run the `lab_03_01.sql` script to create these objects.

```
CREATE TYPE typ_item AS OBJECT  --create object
 (prodid  NUMBER(5),
  price   NUMBER(7,2) )
/
CREATE TYPE typ_item_nst -- define nested table type
  AS TABLE OF typ_item
/
CREATE TABLE pOrder (  -- create database table
     ordid NUMBER(5),
     supplier    NUMBER(5),
     requester   NUMBER(4),
     ordered     DATE,
     items typ_item_nst)
     NESTED TABLE items STORE AS item_stor_tab
/
```

2. The code shown below generates an error. Run the `lab_03_02.sql` script to generate and view the error.

```
BEGIN
  -- Insert an order
  INSERT INTO pOrder
    (ordid, supplier, requester, ordered, items)
     VALUES (1000, 12345, 9876, SYSDATE, NULL);
  -- insert the items for the order created
  INSERT INTO THE (SELECT items
                   FROM   pOrder
                   WHERE  ordid = 1000)
    VALUES(typ_item(99, 129.00));
END;
/
```

Why is the error occurring?

How can you fix the error?

Practice 3 (continued)

Collection Analysis (continued)

3. Examine the following code. This code produces an error. Which line causes the error, and how do you fix it?
   (**Note:** You can run the `lab_03_03.sql` script to view the error output).

```
DECLARE
  TYPE credit_card_typ
  IS VARRAY(100) OF VARCHAR2(30);

  v_mc   credit_card_typ := credit_card_typ();
  v_visa credit_card_typ := credit_card_typ();
  v_am   credit_card_typ;
  v_disc credit_card_typ := credit_card_typ();
  v_dc   credit_card_typ := credit_card_typ();

BEGIN
  v_mc.EXTEND;
  v_visa.EXTEND;
  v_am.EXTEND;
  v_disc.EXTEND;
  v_dc.EXTEND;
END;
/
```

Practice 3 (continued)

Using Collections

In the following practice exercises, you will implement a nested table column in the
CUSTOMERS table and write PL/SQL code to manipulate the nested table.

4. Create a nested table to hold credit card information.

    Create an object type called typ_cr_card. It should have the following specification:
    ```
    card_type   VARCHAR2(25)
    card_num    NUMBER
    ```
    Create a nested table type called typ_cr_card_nst that is a table of typ_cr_card.
    Add a column to the CUSTOMERS table called credit_cards. Make this column a
    nested table of type typ_cr_card_nst. You can use the following syntax:
    ```
    ALTER TABLE customers ADD
    (credit_cards typ_cr_card_nst
        NESTED TABLE credit_cards STORE AD c_c_store_tab);
    ```

5. Create a PL/SQL package that manipulates the credit_cards column in the
    CUSTOMERS table.

    Open the lab_03_05.sql file. It contains the package specification and part of the
    package body. Complete the code so that the package:
    -   Inserts credit card information (the credit card name and number for a specific
        customer.)
    -   Displays credit card information in an unnested format.
    ```
    CREATE OR REPLACE PACKAGE credit_card_pkg
    IS
      PROCEDURE update_card_info
        (p_cust_id NUMBER, p_card_type VARCHAR2, p_card_no
      VARCHAR2);
      PROCEDURE display_card_info
        (p_cust_id NUMBER);
    END credit_card_pkg;  -- package spec
    /
    ```

Using Collections (continued)

```
CREATE OR REPLACE PACKAGE BODY credit_card_pkg
IS

  PROCEDURE update_card_info
    (p_cust_id NUMBER, p_card_type VARCHAR2, p_card_no
VARCHAR2)
  IS
    v_card_info typ_cr_card_nst;
    i INTEGER;
  BEGIN
    SELECT credit_cards
      INTO v_card_info
      FROM customers
      WHERE customer_id = p_cust_id;
    IF v_card_info.EXISTS(1) THEN
-- cards exist, add more

-- fill in code here

    ELSE -- no cards for this customer, construct one

-- fill in code here

    END IF;
  END update_card_info;


  PROCEDURE display_card_info
    (p_cust_id NUMBER)
  IS
    v_card_info typ_cr_card_nst;
    i INTEGER;
  BEGIN
    SELECT credit_cards
      INTO v_card_info
      FROM customers
      WHERE customer_id = p_cust_id;

-- fill in code here to display the nested table
-- contents

  END display_card_info;
END credit_card_pkg;  -- package body
/
```

Practice 3 (continued)

Using Collections (continued)

6. Test your package with the following statements and output:

```
EXECUTE credit_card_pkg.display_card_info(120)
Customer has no credit cards.
PL/SQL procedure successfully completed.

EXECUTE credit_card_pkg.update_card_info -
    (120, 'Visa', 11111111)
PL/SQL procedure successfully completed.

SELECT credit_cards
FROM   customers
WHERE  customer_id = 120;

CREDIT_CARDS(CARD_TYPE, CARD_NUM)
----------------------------------------------------
   TYP_CR_CARD_NST(TYP_CR_CARD('Visa', 11111111))

EXECUTE credit_card_pkg.display_card_info(120)
Card Type: Visa / Card No: 11111111
PL/SQL procedure successfully completed.

EXECUTE credit_card_pkg.update_card_info -
    (120, 'MC', 2323232323)
PL/SQL procedure successfully completed.

EXECUTE credit_card_pkg.update_card_info -
    (120, 'DC', 4444444)
PL/SQL procedure successfully completed.

EXECUTE credit_card_pkg.display_card_info(120)
Card Type: Visa / Card No: 11111111
Card Type: MC / Card No: 2323232323
Card Type: DC / Card No: 4444444
PL/SQL procedure successfully completed.
```

Practice 3 (continued)

Using Collections (continued)

7. Write a `SELECT` statement against the `credit_cards` column to unnest the data. Use the `TABLE` expression.

   For example, if the `SELECT` statement returns:

   ```
   SELECT credit_cards
   FROM   customers
   WHERE  customer_id = 120;

   CREDIT_CARDS(CARD_TYPE, CARD_NUM)
   -----------------------------------------------------------
   TYP_CR_CARD_NST(TYP_CR_CARD('Visa', 11111111),
      TYP_CR_CARD('MC', 2323232323), TYP_CR_CARD('DC', 4444444))
   ```

   then rewrite it using the `TABLE` expression so the results look like:

   ```
   -- Use the table expression so that the result is:
   CUSTOMER_ID CUST_LAST_NAME  CARD_TYPE      CARD_NUM
   ----------- --------------- ------------- -----------
           120 Higgins         Visa             11111111
           120 Higgins         MC             2323232323
           120 Higgins         DC                4444444
   ```

Advanced Interface Methods

4

Oracle University and En-Sof Informatica E Treinamento Ltda use only

Objectives

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

- **Execute external C programs from PL/SQL**
- **Execute Java programs from PL/SQL**

Objectives

In this lesson, you learn how to implement an external C routine from PL/SQL code and how to incorporate Java code into your PL/SQL programs.

Calling External Procedures from PL/SQL

With external procedures, you can make "callouts" and, optionally, "callbacks" through PL/SQL.

DECLARE
● ● ●
BEGIN
● ● ●
EXCEPTION
● ● ●
END;

PL/SQL subprogram

Java class method

C routine

External procedure

Oracle University and En-Sof Informatica E Treinamento Ltda use only
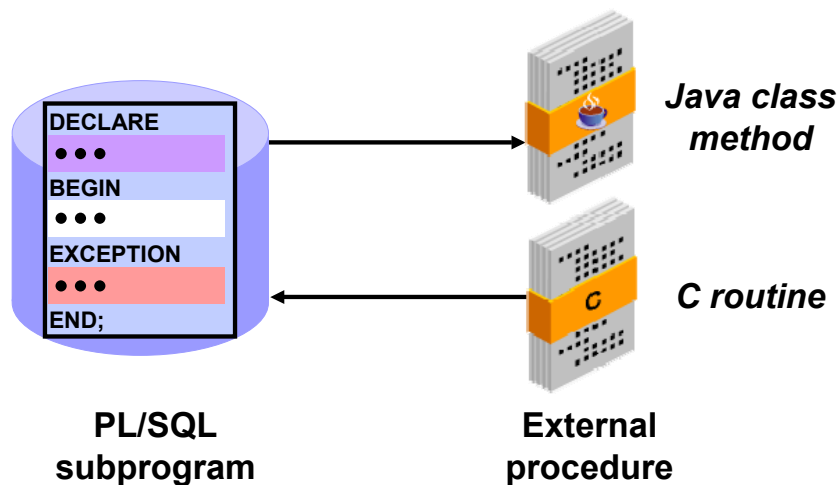
External Procedures: Overview

An *external procedure* (also called an *external routine*) is a routine stored in a dynamic link library (DLL), shared object (`.so` file in UNIX), or libunit in the case of a Java class method, that can perform special purpose processing. You publish the routine with the base language and then call it to perform special purpose processing. You call the external routine from within PL/SQL or SQL. With C, you publish the routine through a library schema object, which is called from PL/SQL, that contains the compiled library file name that is stored on the operating system. With Java, publishing the routine is accomplished through creating a class libunit.

A *callout* is a call to the external procedure from your PL/SQL code.

A *callback* occurs when the external procedure calls back to the database to perform SQL operations. If the external procedure is to execute SQL or PL/SQL, it must "call back" to the database server process to get this work done.

An external procedure enables you to:
- Move computation-bound programs from the client to the server where they execute faster (because they avoid the round trips entailed in across-network communication)
- Interface the database server with external systems and data sources
- Extend the functionality of the database itself

Benefits of External Procedures

- **External procedures integrate the strength and capability of different languages to give transparent access to these routines from within the database.**
- **Extensibility: Provides functionality in the database that is specific to a particular application, company, or technological area**
- **Reusability: Can be shared by all users on a database, as well as moved to other databases or computers, providing standard functionality with limited cost in development, maintenance, and deployment**

ORACLE

Benefits of External Procedures

If you use the external procedure call, you can invoke an external routine by using a PL/SQL program unit. Additionally, you can integrate the powerful programming features of 3GLs with the ease of data access of SQL and PL/SQL commands.

You can extend the database and provide backward compatibility. For example, you can invoke different index or sorting mechanisms as an external procedure to implement data cartridges.

Example

A company has very complicated statistics programs written in C. The customer wants to access the data stored in an Oracle database and pass the data into the C programs. After the execution of the C programs, depending on the result of the evaluations, data is inserted into the appropriate Oracle database tables.

External C Procedure Components
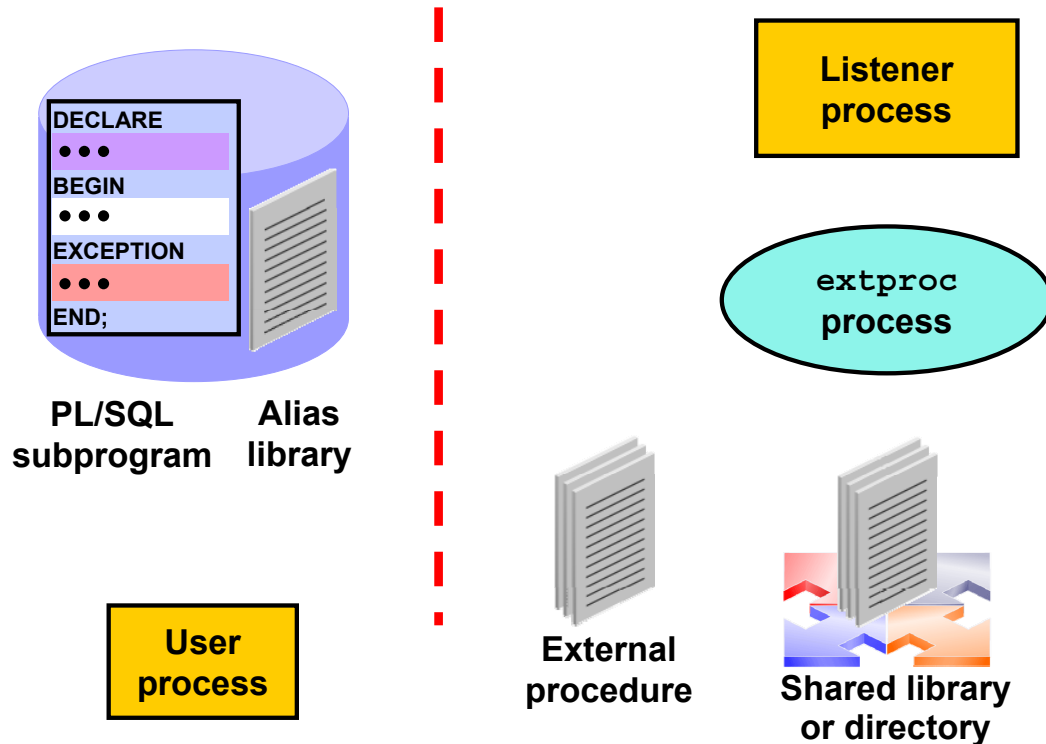
External C Procedure Components

- **External procedure:** A unit of code written in C
- **Shared library:** An operating system file that stores the external procedure
- **Alias library:** A schema object that represents the operating system shared library
- **PL/SQL subprograms:** Packages, procedures, or functions that define the program unit specification and mapping to the PL/SQL library
- **`extproc` process:** A session-specific process that executes external procedures
- **Listener process:** A process that starts the `extproc` process and assigns it to the process executing the PL/SQL subprogram

How PL/SQL Calls a C External Procedure

PL/SQL subprogram

Alias library

BEGIN myproc

User process

DECLARE ••• BEGIN ••• EXCEPTION ••• END;

Listener process

extproc process

External procedure

Shared library
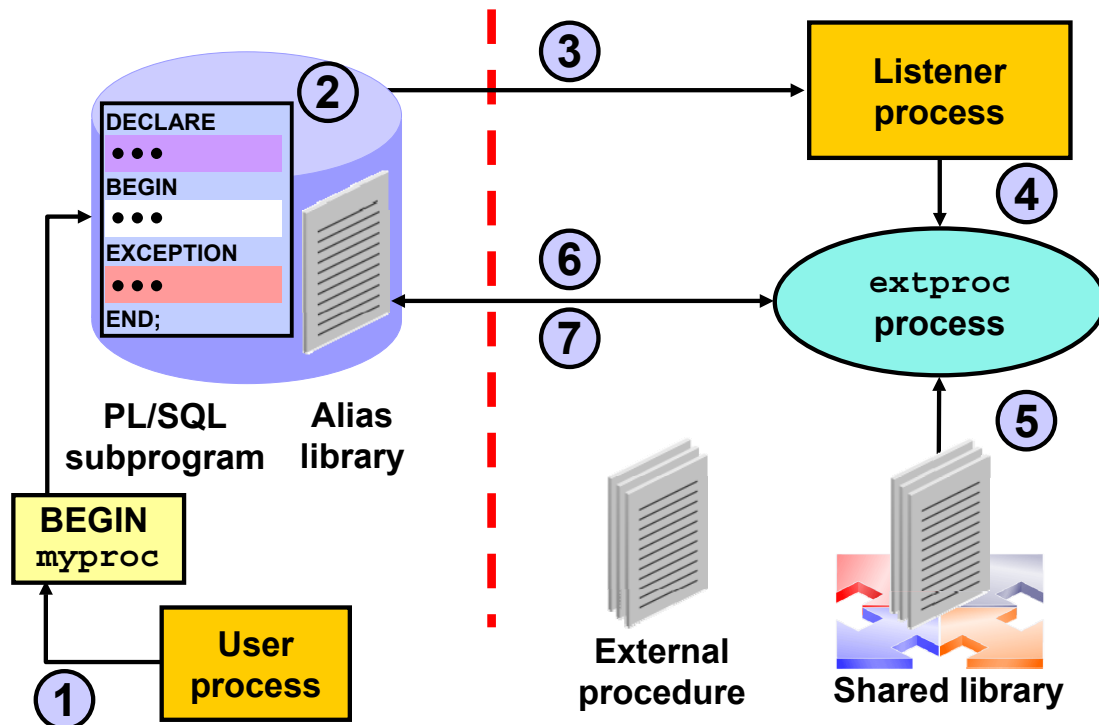
How an External C Procedure Is Called

1. The user process invokes a PL/SQL program.
2. The server process executes a PL/SQL subprogram, which looks up the alias library.
3. The PL/SQL subprogram passes the request to the listener.
4. The listener process spawns the `extproc` process. The `extproc` process remains active throughout your Oracle session until you log off.
5. The `extproc` process loads the shared library.
6. The `extproc` process links the server to the external file and executes the external procedure.
7. The data and status are returned to the server.

Oracle University and En-Sof Informatica E Treinamento Ltda use only

The `extproc` Process

- **The `extproc` process services the execution of external procedures for the duration of the session until the user logs off.**
- **Each session uses a different `extproc` process to execute external procedures.**
- **The listener must be configured to allow the server to be associated to the `extproc` process.**
- **The listener must be on the same machine as the server.**

The `extproc` Process

The `extproc` process performs the following actions:

- **Converts PL/SQL calls to C calls:**
    - Loads the dynamic library
- **Executes the external procedures:**
    - Raises exceptions if necessary
    - Converts C back to PL/SQL
    - Sends arguments or exceptions back to the server process

The Listener Process
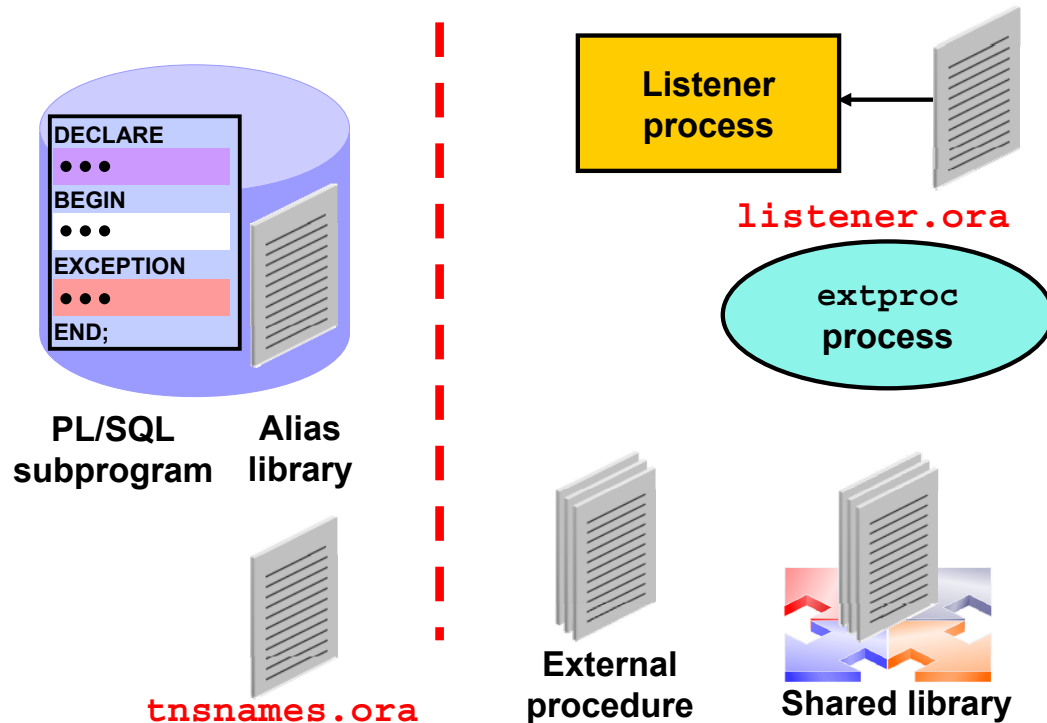
The Listener Process

When the Oracle server executes the external procedure, the request is passed to the listener process, which spawns an `extproc` process that executes the call to the external procedure.

This listener returns the information to the server process. A single `extproc` process is created for each session. The listener process starts the `extproc` process. The external procedure resides in a dynamic library. The Oracle Server 10*g* runs the `extproc` process to load the dynamic library and to execute the external procedure.

3GL Call Dependencies: Example

Libraries are objects with the following dependencies. Given library L1 and procedure P1, which depends on L1, when procedure P1 is executed, library L1 is loaded, and the corresponding external library is dynamically loaded. P1 can now use the external library handle and call the appropriate external functions.

If L1 is dropped, then P1 is invalidated and needs to be recompiled.

Development Steps for External C Procedures

1. **Create and compile the external procedure in 3GL.**

2. **Link the external procedure with the shared library at the operating system level.**

3. **Create an alias library schema object to map to the operating system's shared library.**

4. **Grant execute privileges on the library.**

5. **Publish the external C procedure by creating the PL/SQL subprogram unit specification, which references the alias library.**

6. **Execute the PL/SQL subprogram that invokes the external procedure.**

Development Steps for External C Procedures

Steps 1 and 2 will vary according to the operating system. Consult your operating system or the compiler documentation. After those steps are completed, you need to create an alias library schema object that identifies the operating system's shared library within the server. Any user who needs to execute the C procedure requires execute privileges on the library. Within your PL/SQL code, you map the C arguments to PL/SQL parameters, and lastly, execute your PL/SQL subprogram that invokes the external routine.

Development Steps for External C Procedures

1. ***Varies for each operating system; consult documentation.***

2. **Use the `CREATE LIBRARY` statement to create an alias library object.**

```
CREATE OR REPLACE LIBRARY library_name IS|AS
'file_path';
```

3. **Grant the `EXECUTE` privilege on the alias library.**

```
GRANT EXECUTE ON library_name TO
user|ROLE|PUBLIC;
```

Create the Alias Library

An alias library is a database object that is used to map to an external shared library. Any external procedure that you want to use needs to be stored in a dynamic link library (DLL) or shared object library (SO) operating system file. The DBA controls access to the DLL or SO files by using the `CREATE LIBRARY` statement to create a schema object called an alias library, that represents the external file. The DBA needs to give you `EXECUTE` privileges on the library object so that you can publish the external procedure and then call it from a PL/SQL program.

Steps

1, 2. Steps 1 and 2 will vary for each operating system. Consult your operating system or the compiler documentation.

3. Create an alias library object by using the `CREATE LIBRARY` command:

```
CREATE OR REPLACE LIBRARY c_utility
AS '$ORACLE_HOME/bin/calc_tax.so';
```

The example shows the creation of a database object called `c_utility`, which references the location of the file and the name of the operating system file, `calc_tax.so`.

Create the Alias Library (continued)

4. Grant the `EXECUTE` privilege on the library object:

    ```
    SQL> GRANT EXECUTE ON c_utility TO OE;
    ```

5. Publish the external C routine.
6. Call the external C routine from PL/SQL.

Dictionary Information

The alias library definitions are stored in the `USER_LIBRARIES` and `ALL_LIBRARIES` data dictionary views.

Development Steps for External C Procedures

Publish the external procedure in PL/SQL through call specifications:

- **The body of the subprogram contains the external routine registration.**
- **The external procedure runs on the same machine.**
- **Access is controlled through the alias library.**

External routine within the procedure

Library

BEGIN myproc
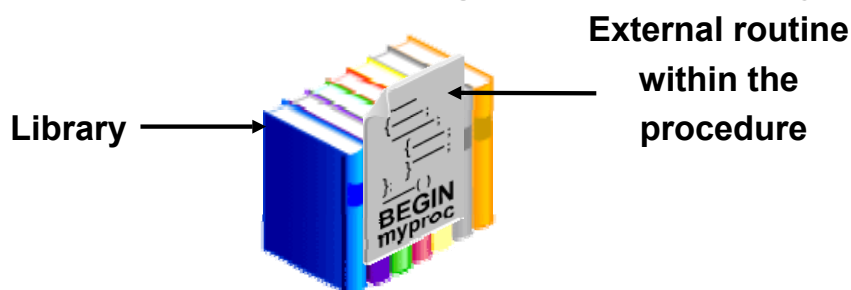
Method to Access a Shared Library Through PL/SQL

You can access a shared library by specifying the alias library in a PL/SQL subprogram. The PL/SQL subprogram then calls the alias library.

- The body of the subprogram contains the external procedure registration.
- The external procedure runs on the same machine.
- Access is controlled through the alias library.

You can publish the external procedure in PL/SQL by:

- Identifying the characteristics of the C procedure to the PL/SQL program
- Accessing the library through PL/SQL

The package specification does not require any changes. You do not need to have definitions for the external procedure.

The Call Specification

Call specifications enable:

- **Dispatching the appropriate C or Java target procedure**
- **Data type conversions**
- **Parameter mode mappings**
- **Automatic memory allocation and cleanup**
- **Purity constraints to be specified, where necessary, for packaged functions that are called from SQL**
- **Calling Java methods or C procedures from database triggers**
- **Location flexibility**

ORACLE

The Call Specification

The current way to publish external procedures is through call specifications. The call specification enables you to call external routines from other languages. Although the specification is designed for intercommunication between SQL, PL/SQL, C, and Java, it is accessible from any base language that can call these languages.

To use an already existing program as an external procedure, load, publish, and then call it.

Call specifications can be specified in any of the following locations:
- Stand-alone PL/SQL procedures and functions
- PL/SQL package specifications
- PL/SQL package bodies
- Object type specifications
- Object type bodies

Note: For functions that already have the pragma RESTRICT_REFERENCES, use the TRUST option. The SQL engine cannot analyze those programs to determine if they are free from side effects. The TRUST option makes it easier to call the Java and C procedures.

The Call Specification

- **Identify the external body within a PL/SQL program to publish the external C procedure.**

```
CREATE OR REPLACE FUNCTION function_name
(parameter_list)
RETURN datatype
  regularbody | externalbody
END;
```

- **The external body contains the external C procedure information.**

```
IS|AS LANGUAGE C
LIBRARY libname
[NAME C_function_name]
[CALLING STANDARD C | PASCAL]
[WITH CONTEXT]
[PARAMETERS (param_1, [param_n]);
```

Publishing an External C Routine

You create the PL/SQL procedure or function and use the IS|AS LANGUAGE C to publish the external C procedure. The external body contains the external routine information.

Syntax Definitions

where:		
	LANGUAGE	The language in which the external routine was written (defaults to C)
	LIBRARY *libname*	Name of the library database object
	NAME *"C_function_name"*	Represents the name of the C function; if omitted, the external procedure name must match the name of the PL/SQL subprogram
	CALLING STANDARD	Specifies the Windows NT calling standard (C or Pascal) under which the external routine was compiled (defaults to C)
	WITH CONTEXT	Specifies that a context pointer will be passed to the external routine for
	parameters	How arguments are passed to the external routine

The Call Specification

- ## The parameter list:

```
parameter_list_element
[ , parameter_list_element ]
```

- ## The parameter list element:

```
{ formal_parameter_name [indicator]
| RETURN INDICATOR
| CONTEXT }
[BY REFERENCE]
[external_datatype]
```

The PARAMETER Clause

The foreign parameter list can be used to specify the position and the types of arguments, as well as indicating whether they should be passed by value or by reference.

Syntax Definitions

where:		
	`formal_parameter_name [INDICATOR]`	Name of the PL/SQL parameter that is being passed to the external routine; the `INDICATOR` keyword is used to map a C parameter whose value indicates whether the PL/SQL parameter is null
	`RETURN INDICATOR`	Corresponds to the C parameter that returns a null indicator for the function
	`CONTEXT`	Specifies that a context pointer will be passed to the external routine
	`BY REFERENCE`	In C, you can pass `IN` scalar parameters by value (the value is passed) or by reference (a pointer to the value is passed). Use `BY REFERENCE` to pass the parameter by reference.
	`External_datatype`	The external data type that maps to a C data type

Note: The `PARAMETER` clause is optional if the mapping of the parameters is done on a positional basis, and indicators, reference, and context are not needed.

Publishing an External C Routine

Example

- **Publish a C function called `c_tax` from a PL/SQL function.**

```
CREATE FUNCTION tax_amt (
 x BINARY_INTEGER)
 RETURN BINARY_INTEGER
  AS LANGUAGE C
  LIBRARY c_utility
  NAME "c_tax";
/
```

- **The C prototype:**

```
int c_tax (int x_val);
```

Example

You have an external C function called `c_tax` that takes in one argument, the total sales amount. The function returns the tax amount calculated at 8%. The prototype for your `c_tax` function follows:

```
int c_tax (int x_val);
```

To publish the `c_tax` function in a stored PL/SQL function, use the `AS LANGUAGE C` clause within the function definition. The `NAME` identifies the name of the C function. Double quotation marks are used to preserve the case of the function defined in the C program. The `LIBRARY` identifies the library object that locates where the C file is. The `PARAMETERS` clause is not needed in this example because the mapping of the parameters is done on a positional basis.

Executing the External Procedure

1. **Create and compile the external procedure in 3GL.**
2. **Link the external procedure with the shared library at the operating system level.**
3. **Create an alias library schema object to map to the operating system's shared library.**
4. **Grant execute privileges on the library.**
5. **Publish the external C procedure by creating the PL/SQL subprogram unit specification, which references the alias library.**
6. **Execute the PL/SQL subprogram that invokes the external procedure.**

Executing the External Procedure: Example

Call the external C procedure within a PL/SQL block:

```
DECLARE
  CURSOR cur_orders IS
    SELECT order_id, order_total
    FROM   orders;
  v_tax  NUMBER(8,2);
BEGIN
  FOR order_record IN cur_orders
  LOOP
    v_tax := tax_amt(order_record.order_total);
    DBMS_OUTPUT.PUT_LINE('Total tax: ' || v_tax);
  END LOOP;
END;
```

Overview of Java

The Oracle database can store Java classes and Java source, which:

- **Are stored in the database as procedures, functions, or triggers**
- **Run inside the database**
- **Manipulate data**
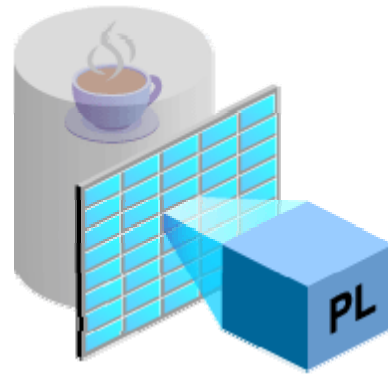
Java Overview

The Oracle database can store Java classes (`.class` files) and Java source code (`.java` files) and execute them inside the database, as stored procedures and triggers. These classes can manipulate data, but cannot display GUI elements such as AWT or Swing components. Running Java inside the database helps these Java classes to be called many times and manipulate large amounts of data, without the processing and network overhead that comes with running on the client machine.

You must write these named blocks and then define them by using the `loadjava` command or the SQL `CREATE FUNCTION`, `CREATE PROCEDURE`, `CREATE TRIGGER`, or `CREATE PACKAGE` statements.

How PL/SQL Calls a Java Class Method

libunits
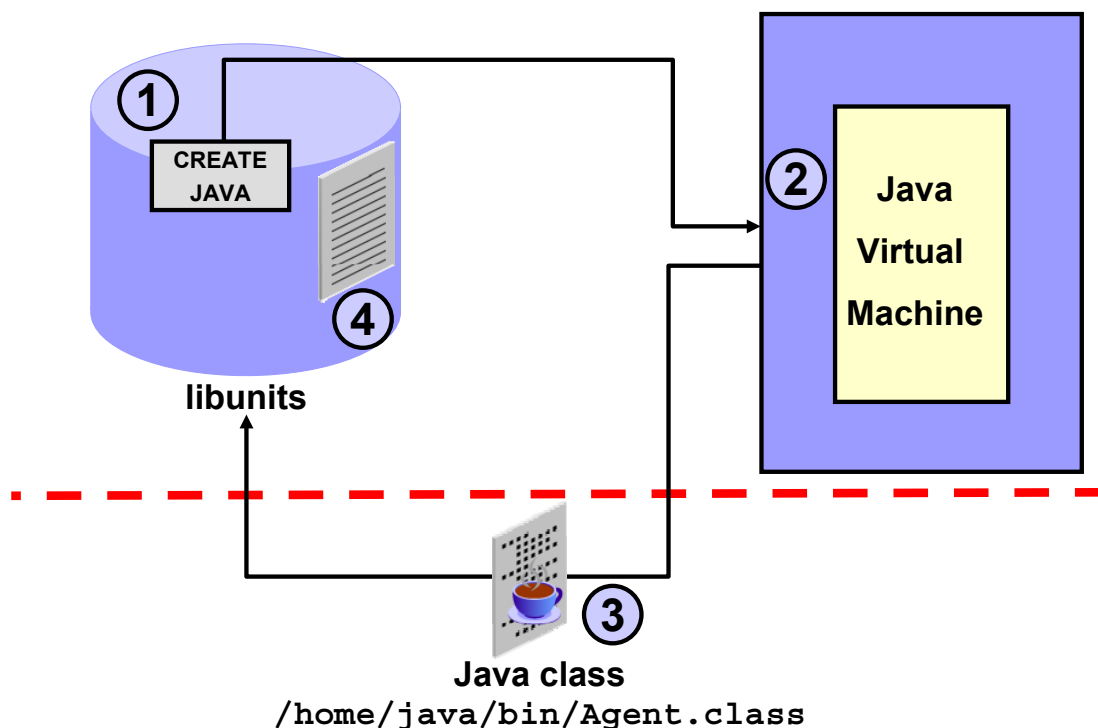
Java class
`/home/java/bin/Agent.class`

Calling a Java Class Method Using PL/SQL

The command-line utility `loadjava` uploads Java binaries and resources into a system-generated database table. It then uses the `CREATE JAVA` statement to load the Java files into RDBMS libunits. You can upload Java files from file systems, Java IDEs, intranets, or the Internet.

When the `CREATE JAVA` statement is invoked, the Java Virtual Machine library manager on the server loads Java binaries and resources from local `BFILE`s or `LOB` columns into RDBMS libunits. Libunits can be considered analogous to DLLs written in C, although they map one-to-one with Java classes, whereas DLLs can contain more than one routine.

Development Steps for Java Class Methods
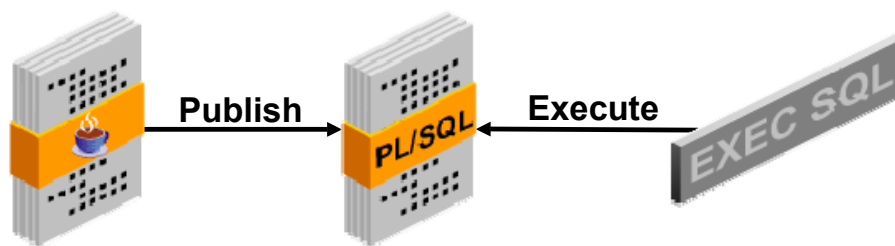
1. **Upload the Java file.**
2. **Publish the Java class method by creating the PL/SQL subprogram unit specification that references the Java class methods.**
3. **Execute the PL/SQL subprogram that invokes the Java class method.**

Steps for Using Java Class Methods

Similar to using external C routines, the following steps are required to complete the setup before executing the Java class method from PL/SQL.

1. Upload the Java file. This takes an external Java binary file and stores the Java code in the database.
2. Publish the Java class method by creating the PL/SQL subprogram unit specification that references the Java class methods.
3. Execute the PL/SQL subprogram that invokes the Java class method.

Loading Java Class Methods

1. **Upload the Java file.**
   - **At the operating system, use the `loadjava` command-line utility to load either the Java class file or the Java source file.**
- **To load the Java class file, use:**

```
>loadjava –user oe/oe Factorial.class
```

- **To load the Java source file, use:**

```
>loadjava –user oe/oe Factorial.java
```

   - **If you load the Java source file, you do not need to load the Java class file.**

Loading Java Class Methods

Java classes and their methods are stored in RDBMS libunits in which you can load Java sources, binaries, and resources.

Use the `loadjava` command-line utility to load and resolve the Java classes. Using the `loadjava` utility, you can upload a Java source, class, and resource files into an Oracle database, where they are stored as Java schema objects. You can run `loadjava` from the command line or from an application.

After the file is loaded, it is visible in the data dictionary views.

```
SELECT object_name, object_type FROM   user_objects
WHERE  object_type like 'J%';
OBJECT_NAME                     OBJECT_TYPE
------------------------------- -------------------------
Factorial                       JAVA CLASS
SELECT text FROM   user_source WHERE name = 'Factorial';
TEXT
--------------------------------------------------------
public class Factorial {
  public static int calcFactorial (int n) {
    if (n == 1) return 1;
    else return n * calcFactorial (n - 1) ;
  }}
```

Publishing a Java Class Method

2. **Publish the Java class method by creating the PL/SQL subprogram unit specification that references the Java class methods.**
   - **Identify the external body within a PL/SQL program to publish the Java class method.**
   - **The external body contains the name of the Java class method.**

```
CREATE OR REPLACE
{   PROCEDURE procedure_name [(parameter_list)]
  | FUNCTION function_name [(parameter_list]...)]
    RETURN datatype}
    regularbody | externalbody
END;
```

```
{IS | AS} LANGUAGE JAVA
  NAME 'method_fullname (java_type_fullname
      [, java_type_fullname]...)
      [return java_type_fullname]';
```

ORACLE

Publishing a Java Class Method

The publishing of Java class methods is specified in the AS LANGUAGE clause. This call specification identifies the appropriate Java target routine, data type conversions, parameter mode mappings, and purity constraints. You can publish value-returning Java methods as functions and void Java methods as procedures.

Publishing a Java Class Method

- **Example:**

```
CREATE OR REPLACE FUNCTION plstojavafac_fun
(N NUMBER)
 RETURN NUMBER
 AS
    LANGUAGE JAVA
    NAME 'Factorial.calcFactorial
      (int) return int';
```

- **Java method definition:**

```
public class Factorial {
   public static int calcFactorial (int n) {
      if (n == 1) return 1;
      else return n * calcFactorial (n - 1) ;
   }
}
```

Example

If you want to publish a Java method named `calcFactorial` that returns the factorial of its argument, as explained in the preceding example:

- The `NAME` clause string uniquely identifies the Java method
- The PL/SQL function shown corresponds with regard to the parameters
- The parameter named `N` corresponds to the `int` argument

Executing the Java Routine

1. **Upload the Java file.**
2. **Publish the Java class method by creating the PL/SQL subprogram unit specification that references the Java class methods.**
3. **Execute the PL/SQL subprogram that invokes the Java class method.**

Example (continued)

You can call the `calcFactorial` class method by using the following command:

```
EXECUTE DBMS_OUTPUT.PUT_LINE(plstojavafac_fun (5))
120
```

Alternatively, to execute a `SELECT` statement from the `DUAL` table:

```
SELECT plstojavafac_fun (5)
FROM dual;

PLSTOJAVAFAC_FUN(5)
-------------------
                120
```

Creating Packages for Java Class Methods

```
CREATE OR REPLACE PACKAGE Demo_pack
AUTHID DEFINER
AS
  PROCEDURE plsToJ_InSpec_proc
  (x BINARY_INTEGER, y VARCHAR2, z DATE)
END;
```

```
CREATE OR REPLACE PACKAGE BODY Demo_pack
AS
  PROCEDURE plsToJ_InSpec_proc
    (x BINARY_INTEGER, y VARCHAR2, z DATE)
  IS LANGUAGE JAVA
  NAME 'pkg1.class4.J_InSpec_meth
        (int, java.lang.String, java.sql.Date)';
```

Creating Packages for Java Class Methods

These examples create a package specification and body Demo_pack.

The package is a container structure. It defines the specification of the PL/SQL procedure named plsToJ_InSpec_proc.

Note that you cannot tell whether this procedure is implemented by PL/SQL or by way of an external procedure. The details of the implementation appear only in the package body in the declaration of the procedure body.

Summary

In this lesson, you should have learned how to:

- **Use external C routines and call them from your PL/SQL programs**
- **Use Java methods and call them from your PL/SQL programs**

Summary

You can embed calls to external C programs from your PL/SQL programs by publishing the external routines in a PL/SQL block. You can take external Java programs and store them in the database to be called from PL/SQL functions, procedures, and triggers.

Practice Overview

This practice covers the following topics:

- **Writing programs to interact with C routines**
- **Writing programs to interact with Java code**

ORACLE

Practice Overview

In this practice, you will write two PL/SQL programs. One program calls an external C routine and the second program calls a Java routine.

Practice 4

1. An external C routine definition is created for you. The `.c` file is stored in the `$HOME/labs` directory on the database server. This function returns the tax amount based on the total sales figure passed to the function as a parameter. The name of the `.c` file is named `calc_tax.c`. The shared object file name is `calc_tax.so`. The function is defined as:

   ```
   calc_tax(n)
   int n;
   {
     int tax;
     tax=(n*8)/100;
     return(tax);
   }
   ```

   a. Create `calc_tax.so` file by using the following command:
   ```
   cc -shared -o calc_tax.so calc_tax.c
   ```

   b. Copy the file `calc_tax.so` to `$ORACLE_HOME/bin` directory using the following command:
   ```
   cp calc_tax.so $ORACLE_HOME/bin
   ```

   c. Create the library object. Name the library object `c_code` and define its path as:
   ```
   CREATE OR REPLACE LIBRARY c_code
   AS '$ORACLE_HOME/bin/calc_tax.so';
   /
   ```

   d. Publish the external C routine.
   Create a function named `call_c`. This function has one numeric parameter and it returns a binary integer. Identify the `AS LANGUAGE`, `LIBRARY`, and `NAME` clauses of the function.

   e. Create a procedure to call the `call_c` function created in the previous step. Name this procedure `C_OUTPUT`. It has one numeric parameter. Include a `DBMS_OUTPUT.PUT_LINE` statement so that you can view the results returned from your C function.

   f. Set serveroutput ON and execute the `C_OUTPUT` procedure.

Practice 4 (continued)

2. A Java method definition is created for you. The method accepts a 16-digit credit card number as the argument and returns the formatted credit card number (4 digits followed by a space). The name of the .class file is FormatCreditCardNo.class. The method is defined as:

```
public class FormatCreditCardNo
{
public static final void formatCard(String[] cardno)
{
int count=0, space=0;
String oldcc=cardno[0];
String[] newcc= {""};
while (count<16)
{
newcc[0]+= oldcc.charAt(count);
space++;
if (space ==4)
{  newcc[0]+=" "; space=0;  }
count++;
}
cardno[0]=newcc [0];
}
}
```

a. Load the .java source file.

b. Publish the Java class method by defining a PL/SQL procedure named CCFORMAT. This procedure accepts one IN OUT parameter.

Use the following definition for the NAME parameter:
```
NAME 'FormatCreditCardNo.formatCard(java.lang.String[])';
```

c. Execute the Java class method. Define one SQL*Plus variable, initialize it, and use the EXECUTE command to execute the CCFORMAT procedure.

```
EXECUTE ccformat(:x);


PRINT x
                   X
-------------------
1234 5678 1234 5678
```

PL/SQL Server Pages

Objectives

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

- **Embed PL/SQL code in Web pages (PL/SQL server pages)**
- **Explain the format of a PL/SQL server page**
- **Write the code and content for the PL/SQL server page**
- **Load the PL/SQL server page into the database as a stored procedure**
- **Run a PL/SQL server page via a URL**
- **Debug PL/SQL server page problems**

ORACLE

Objectives

In this lesson, you learn about the powerful features of PL/SQL Server Pages (PSP). Using PSP, you can embed PL/SQL in an HTML Web page.

PSP: Uses and Features

- **Uses:**
  - **If you have a large body of HTML, and want to include dynamic content or make it the front end of a database application**
  - **If most work is done using HTML authoring tools**
- **Features:**
  - **You can include JavaScript or other client-side script code in a PL/SQL server page.**
  - **PSP uses the same script tag syntax as JavaServer Pages (JSP), to make it easy to switch back and forth.**
  - **Processing is done on the server.**
  - **The browser receives a plain HTML page with no special script tags.**

ORACLE

PSP Uses and Features

You can produce HTML pages with dynamic content in several ways. One method is to create PSP. This is useful when you have a large body of HTML, and want to include dynamic content or make it the front end of a database application. If most of the work is done through an HTML authoring tool, PSP is more efficient.

You can also use the PL/SQL Web Toolkit to generate PSPs. This toolkit provides packages such as `OWA`, `htp`, and `htf` that are designed for generating Web pages. For more information, take the *Oracle AS 10g: Develop Web Pages with PL/SQL* course. This is useful when there is a large body of PL/SQL code that produces formatted output. If you use authoring tools that produce PL/SQL code for you, such as the page-building wizards in Oracle Application Server Portal, then it might be less convenient to use PSP.

Oracle University and En-Sof Informatica E Treinamento Ltda use only

Format of the PSP File

- **The file must have a `.psp` extension.**
- **The `.psp` file can contain text, tags, PSP directives, declarations, and scriptlets.**
- **Typically, HTML provides the static portion of the page, and PL/SQL provides the dynamic content.**

Test.psp

Format of the PSP File

It is easier to maintain the PSP file if you keep all your directives and declarations together near the beginning of a PL/SQL server page. To share procedures, constants, and types across different PL/SQL server pages, compile them into a separate package in the database by using a plain PL/SQL source file. Although you can reference packaged procedures, constants, and types from PSP scripts, the PSP scripts can only produce stand-alone procedures, not packages.

Page Directive
Specifies characteristics of the PL/SQL server page:
- What scripting language it uses
- What type of information (MIME type) it produces
- What code to run to handle all uncaught exceptions. This might be an HTML file with a friendly message, renamed to a `.psp` file.

  Syntax:
  ```
  <%@ page [language="PL/SQL"]
  contentType="content type string"] [errorPage="file.psp"] %>
  ```

Procedure Directive
Specifies the name of the stored procedure produced by the PSP file. By default, the name is the file name without the `.psp` extension.

  Syntax:
  ```
  <%@ plsql procedure="procedure name" %>
  ```

Format of the PSP File (continued)

Parameter Directive

Specifies the name, and optionally the type and default, for each parameter expected by the PSP stored procedure.

Syntax:
```
<%@ plsql parameter="parameter name"
 [type="PL/SQL type"] [default="value"] %>
```

If the parameter data type is CHARACTER, put single quotation marks around the default value, with double quotation marks surrounding the entire default value.

Include Directive

Specifies the name of a file to be included at a specific point in the PSP file. The file must have an extension other than .psp. It can contain HTML, PSP script elements, or a combination of both. The name resolution and file inclusion happens when the PSP file is loaded into the database as a stored procedure, so any changes to the file after that are not reflected when the stored procedure is run.

Syntax:
```
<%@ include file="path name" %>
```

Declaration Block

Declares a set of PL/SQL variables that are visible throughout the page, not just within the next BEGIN/END block. This element typically spans multiple lines, with individual PL/SQL variable declarations ended by semicolons.

Syntax:
```
<%! PL/SQL declaration; [ PL/SQL declaration; ] ... %>
```

Code Block (Scriptlets)

Executes a set of PL/SQL statements when the stored procedure is run. This element typically spans multiple lines, with individual PL/SQL statements ended by semicolons. The statements can include complete blocks, or can be the bracketing parts of IF/THEN/ELSE or BEGIN/END blocks. When a code block is split into multiple scriptlets, you can put HTML or other directives in the middle, and those pieces are conditionally executed when the stored procedure is run.

Syntax:
```
<% PL/SQL statement; [ PL/SQL statement; ] ... %>
```

Expression Block

Specifies a single PL/SQL expression, such as a string, an arithmetic expression, a function call, or a combination of those things. The result is substituted as a string at that spot in the HTML page that is produced by the stored procedure. You do not need to end the PL/SQL expression with a semicolon.

Syntax:
```
<%= PL/SQL expression %>
```

Note: To identify a file as a PL/SQL server page, include a
`<%@ page language="PL/SQL" %>` directive somewhere in the file. This directive is for compatibility with other scripting environments.

Development Steps for PSP

1. **Create the PSP.**
2. **Load the PSP into the database as a stored procedure.**

```
loadpsp [ -replace ]
-user username/password[@connect_string]
[ include_file_name ... ] [ error_file_name ]
psp_file_name ...
```

3. **Run the PSP through a URL.**

```
http://sitename/schemaname/pspname?parmname1=
value1&parmname2=value2
```

Steps to Create a PSP

Step 1

Create an HTML page, embedding the PL/SQL code in the HTML page.

Development Steps for PSP

Creating the PSP:

```
<%@ page language="PL/SQL" %>          ←——— Page directive
<%@ plsql procedure="show_table" %>    ←——— Procedure directive
<% -- Inventories Table Contents -- %> ←— Comment
<HTML>
<HEAD><TITLE>Show Contents of Inventories </TITLE></HEAD>
<BODY>
<p><b><font face="Arial, Helvetica, Tahoma"
 size="4">INVENTORIES TABLE: </font></b></p>
<p><%
declare                                ←——— Scriptlet
dummy boolean;
begin
dummy := owa_util.tableprint('INVENTORIES','border');
end;
%> </p>
<p> </p><p> </p><p> </p></BODY>
</HTML>
```

Creating the PSP

First, create an HTML page, embedding the PL/SQL code in the HTML page. In this example, the contents of the INVENTORIES table are displayed in a Web page.

The page directive identifies the scripting language. The procedure directive identifies that a procedure named show_table will be created and stored in the database to represent this HTML page. The scriptlet executes a set of PL/SQL statements when the stored procedure is run. The result is substituted as a string at that spot in the HTML page that is produced by the stored procedure. The owa_util.tableprint procedure prints out the contents of a database table that is identified to the procedure through the first parameter.

Note: owa_util.tableprint is part of the PL/SQL Web Toolkit and is installed in the SYS schema.

Include Comments

To put a comment in the HTML portion of a PL/SQL server page, for the benefit of people reading the PSP source code, use the following syntax:

Syntax:
```
<%-- Comment text --%>
```

These comments do not appear in the HTML output from the PSP.

To create a comment that is visible in the HTML output, place the comment in the HTML portion and use the regular HTML comment syntax:

Syntax:
```
<!-- Comment text -->
```

Development Steps for PSP

- **Loading the PSP into the database from the operating system:**

```
>loadpsp –replace –user oe/oe show_table.psp
"show_table.psp" : procedure "show_table" created.
>
```

- **Optionally include other file names and the error file name:**

```
>loadpsp –replace –user oe/oe
banner.inc error.psp show_table.psp
"banner.inc": uploaded.
"error.psp": procedure "error" created.
"show_table.psp" : procedure "show_table" created.
>
```

Loading the PSP

Step 2

In the second step, you load one or more PSP files into the database as stored procedures. Each `.psp` file corresponds to one stored procedure. To perform a "CREATE OR REPLACE" on the stored procedures, include the `-replace` flag.

The loader logs on to the database using the specified username, password, and connect string. The stored procedures are created in the corresponding schema.

In the first example:
- The stored procedure is created in the database. The database is accessed as user `oe` with password `oe`, both when the stored procedure is created and when it is executed.
- `show_table.psp` contains the main code and text for the Web page.

In the second example:
- The stored procedure is created in the database. The database is accessed as user `oe` with password `oe`, both to create the stored procedure and when the stored procedure is executed.
- `banner.inc` is a file containing boilerplate text and script code, that is included by the `.psp` file. The inclusion happens when the PSP is loaded into the database, not when the stored procedure is executed.
- `error.psp` is a file containing code or text that is processed when an unhandled exception occurs, to present a friendly page rather than an internal error message.
- `show_table.psp` contains the main code and text for the Web page.

Loading the PSP (continued)

Include the names of all the include files (whose names do not have the `.psp` extension) before the names of the PL/SQL server pages (whose names have the `.psp` extension). Also include the name of the file specified in the `errorPage` attribute of the page directive. These file names on the `loadpsp` command line must exactly match the names specified within the PSP include and page directives, including any relative pathname such as `../include/`.

Development Steps for PSP

The `show_table` procedure is stored in the data dictionary views.

```
SQL> SELECT text
  2  FROM   user_source
  3  WHERE  name = 'SHOW_TABLE';

TEXT
----------------------------------------------------------------
PROCEDURE show_table  AS
 BEGIN NULL;
...
declare
dummy boolean;
begin
dummy := owa_util.tableprint('INVENTORIES','border');
end;
...
23 rows selected.
```
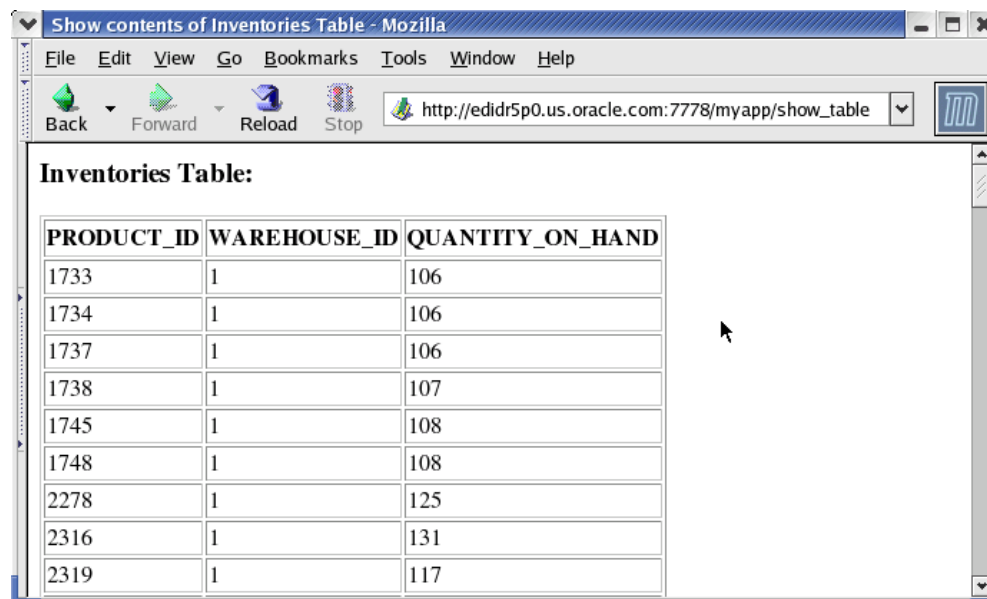
Loading the PSP (continued)

After the `loadpsp` utility is run, the procedure is created and stored in the database.

Development Steps for PSP

Running the PSP through a URL:

Running the PSP

Step 3

For the third step, run the PSP in a browser. Identify the HTTP URL through a Web browser or some other Internet-aware client program. The virtual path in the URL depends on the way the Web gateway is configured. The name of the stored procedure is placed at the end of the virtual path.

Using `METHOD=GET`, the URL may look like this:

```
http://sitename/DAD/pspname?parmname1=value1&parmname2=value2
```

Using `METHOD=POST`, the URL does not show the parameters:

```
http://sitename/DAD/pspname
```

The `METHOD=GET` format is more convenient for debugging and allows visitors to pass exactly the same parameters when they return to the page through a bookmark.

The `METHOD=POST` format allows a larger volume of parameter data, and is suitable for passing sensitive information that should not be displayed in the URL.

Printing the Table Using a Loop

- **To print the results of a multirow query, use a loop:**

```
<% FOR item IN (SELECT * FROM some_table) LOOP %>
  <TR>
  <TD><%= item.col1 %></TD>
  <TD><%= item.col2 %></TD>
  </TR>
<% END LOOP; %>
```

- **Alternatively, use** `OWA_UTIL.TABLEPRINT` **or** `OWA_UTIL.CELLSPRINT` **procedures from the PL/SQL Web Toolkit.**

ORACLE

Printing the Content of a Table

You can iterate through each row of the result set, printing the appropriate columns using HTML list or table tags. Following is an example of a list:

```
<%@ page language="PL/SQL" %>
<%@ plsql procedure="show_customers" %>
<HTML>
<HEAD><TITLE>Show Contents of Customers (using a loop)
</TITLE></HEAD>
<BODY>
<UL>
<% for item in (select customer_id, cust_first_name,
                credit_limit, cust_email
                from customers order by credit_limit) loop %>
<LI>
ID = <%= item.customer_id %><BR>
Name = <%= item.cust_first_name %><BR>
Credit = <%= item.credit_limit %><BR>
Email = <I><%= item.cust_email %></I><BR>
<% end loop; %>
</UL>
</BODY>
</HTML>
```

Specifying a Parameter

- **Include the parameter directive in the `.psp` file.**
  - **Syntax:**

```
<%@ plsql parameter="parameter name"
  [type="PL/SQL type"] [default="value"] %>
```

  - **Example:**

```
<%@ plsql parameter="mincredit" type="NUMBER"
default="3000" %>
```

- **Assign the parameter a value through the URL call:**

```
http://edidr5p0.us.oracle.com/DAD
/show_customers_hc?mincredit=3000
```

Specifying a Parameter
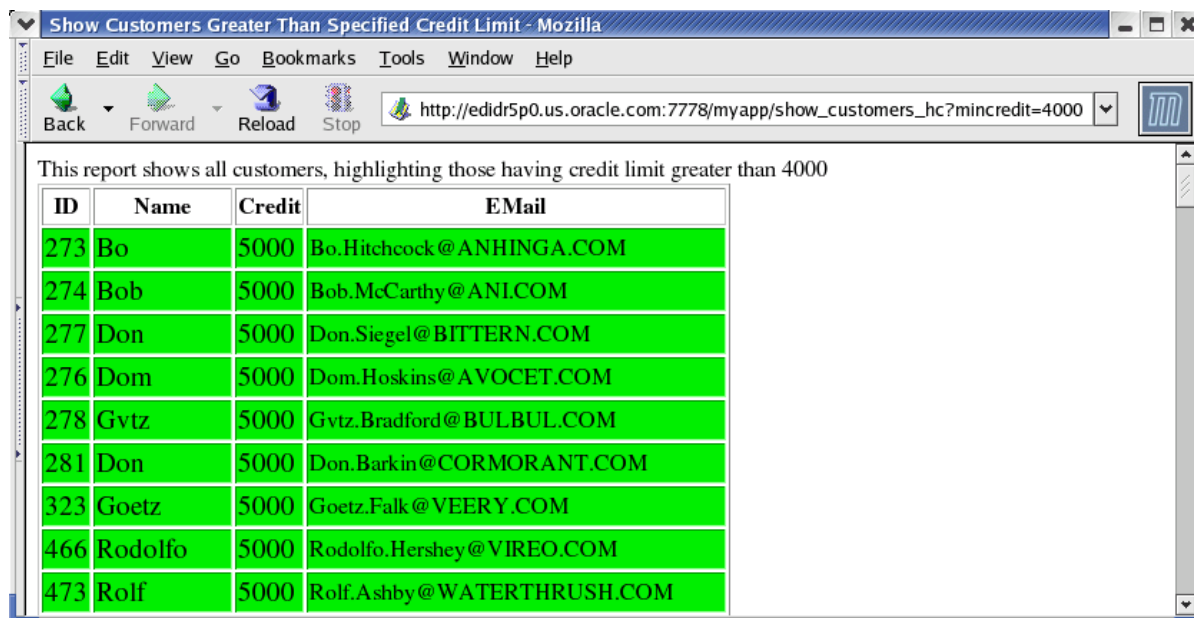
You can pass parameters to the PSP by identifying the parameter name and value in the URL call.

Specifying a Parameter (continued)

The example below creates a parameter named `mincredit`. There is also some conditional processing to highlight values that are greater than a specified price.

```
<%@ page language="PL/SQL" %>
<%@ plsql procedure="show_customers_hc" %>
<%@ plsql parameter="mincredit" type="NUMBER" default="3000"
%>
<%! color varchar2(7); %>
<HTML>
<HEAD><TITLE>Show Customers Greater Than Specified Credit
Limit</TITLE></HEAD>
<BODY>
<P>This report shows all customers, highlighting those having
credit limit is greater than <%= mincredit %>.
<TABLE BORDER>
<TR>
<TH>ID</TH>
<TH>Name</TH>
<TH>Credit</TH>
<TH>Email  </TH>
</TR>
<%
for item in (select * from customers
                order by credit_limit desc) loop
  if item.credit_limit > mincredit then
    color := '#white';
  else
    color := '#green';
  end if;
%>
<TR BGCOLOR="<%= color %>">
<TD><BIG><%= item.customer_id %></BIG></TD>
<TD><BIG><%= item.cust_first_name %></BIG></TD>
<TD><BIG><%= item.credit_limit %></BIG></TD>
<TD><%= item.cust_email %></TD>
</TR>
<% end loop; %>
</TABLE>
</BODY>
</HTML>
```

Specifying a Parameter
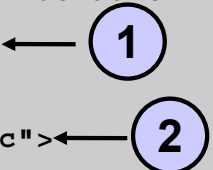
Specifying a Parameter (continued)

You passed `mincredit=4000` as the parameter along with the URL. The output shows all the records and highlights those having a credit limit greater than 4,000.

Using an HTML Form to Call a PSP

1. **Create an HTML form.**
2. **Call the PSP from the form.**

```
<%@ page language="PL/SQL" %>
<%@ plsql procedure="show_customer_call" %>
<%@ plsql parameter="mincredit" type="NUMBER" default=
"3000" %>                                            ①
<html>
<body>
<form method="POST" action="show_customers_hc">    ②
<p>Enter the credit limit:
<input type="text" name="mincredit">
<input type="submit" value="Submit">
</form>
</body>
</html>
```
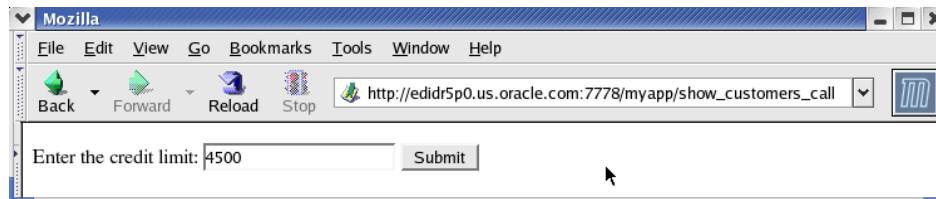
ORACLE

Calling a PSP from an HTML Form

Create an HTML form that calls the PSP. To avoid coding the entire URL of the stored procedure in the `ACTION=` attribute of the form, make the form a PSP file so that it goes in the same directory as the PSP file it calls.

Using an HTML Form to Call a PSP

Calling a PSP from an HTML Form (continued)

Initially, you are calling the HTML form that accepts the credit limit from the user. After submitting the HTML form, call the PSP, which shows all the records and highlight all the records having a credit limit greater than the value submitted by the user.

Debugging PSP Problems

- **Code the PL/SQL syntax and PSP directive syntax correctly. It will not compile with syntax errors.**
- **Run the PSP file by requesting its URL in a Web browser. An error might indicate that the file is not found.**
- **When the PSP script is run, and the results come back to the browser, use standard debugging techniques to check for and correct wrong output.**
- **Use `htp.p('string')` to print information to the screen.**

Debugging PSP Problems

The first step is to code PL/SQL syntax and PSP directive syntax correctly. It will not compile with syntax errors.

- Use semicolons to terminate lines if required.
- If required, enclose a value with quotation marks. You may need to enclose a value that is within single quotation marks (needed by PL/SQL) inside double quotation marks (needed by PSP).
- Mistakes in the PSP directives are usually reported through PL/SQL syntax messages. Check that your directives use the right syntax, that directives are closed properly, and that you are using the right element (declaration, expression, or code block) depending on what goes inside it.
- PSP attribute names are case sensitive. Most are specified in all lowercase; `contentType` and `errorPage` must be specified as mixed-case.

Run the PSP file by requesting its URL in a Web browser.

- Request the right virtual path, depending on the way the Web gateway is configured. Typically, the path includes the host name, optionally a port number, the schema name, and the name of the stored procedure (with no `.psp` extension).
- If you use the `-replace` option when compiling the file, the old version of the stored procedure is erased. You may want to test new scripts in a separate schema until they are ready, then load them into the production schema.
- If you copied the file from another file, remember to change any procedure name directives in the source to match the new file name.

Debugging PSP Problems (continued)

- If you receive one file-not-found error, make sure to request the latest version of the page the next time. The error page may be cached by the browser. You may need to press [Shift] and click Reload in the browser to bypass its cache.

When the PSP script is run, and the results come back to the browser, use standard debugging techniques to check for and correct wrong output. The tricky part is to set up the interface between different HTML forms, scripts, and CGI programs so that all the right values are passed into your page. The page may return an error because of a parameter mismatch.

- To see exactly what is being passed to your page, use METHOD=GET in the calling form so that the parameters are visible in the URL.
- Make sure that the form or CGI program that calls your page passes the correct number of parameters, and that the names specified by the NAME=attributes on the form match the parameter names in the PSP file. If the form includes any hidden input fields, or uses the NAME= attribute on the Submit or Reset buttons, then the PSP file must declare equivalent parameters.
- Make sure that the parameters can be cast from string into the correct PL/SQL types. For example, do not include alphabetic characters if the parameter in the PSP file is declared as a NUMBER.
- Make sure that the query string of the URL consists of name-value pairs, separated by equal signs, especially if you are passing parameters by constructing a hard-coded link to the page.
- If you are passing a lot of parameter data, such as large strings, you may exceed the volume that can be passed with METHOD=GET. You can switch to METHOD=POST in the calling form without changing your PSP file.
- You can display text or variables by putting the following in your code:

```
htp.p(' My Var: ' || my_var);
```

When you run the program, the information is displayed on the screen.

Summary

In this lesson, you should have learned how to:

- **Define PL/SQL server pages**
- **Explain the format of a PL/SQL server page**
- **Write the code and content for the PL/SQL server page**
- **Load the PL/SQL server page into the database as a stored procedure**
- **Run a PL/SQL server page via a URL**
- **Debug PL/SQL server page problems**

ORACLE

Summary

You can use PL/SQL embedded in HTML and store the code as a PL/SQL server page (PSP) in the database. The three steps for creating a PSP are:

1. Create the PSP.
2. Load the PSP into the database as a stored procedure.
3. Run the PSP in a browser.

Practice Overview

This practice covers the following topics:

- **Creating a PSP**
- **Loading a PSP**
- **Running the PSP through the browser**

ORACLE

Practice Overview

In this practice, you write and deploy a PSP that retrieves order information. You will also write and deploy a PSP that retrieves customer information where Customer ID is passed as a parameter.

Practice 5

Note: The instructor needs to set up a DAD for the class.

1. Create a PL/SQL server page to display order information. Name the procedure as `show_orders`. Display the following fields:

   ```
   ORDER_ID
   ORDER_MODE
   CUSTOMER_ID
   ORDER_STATUS
   ORDER_TOTAL
   TAX
   SALES_REP_ID
   ```

   Note: TAX should be displayed using the `calc_c` function created in Lesson 4.

   a. Use the `lab_05_01.psp` file containing the HTML code. After creating the PSP, load it from the operating system

   b. Request the `show_orders` PSP from your browser.

2. Create a PL/SQL server page to display the following customer information:

   ```
   CUSTOMER_ID
   CUST_FIRST_NAME
   CUST_LAST_NAME
   CREDIT_LIMIT
   CUST_EMAIL
   ```

   a. Use the `lab_05_02a.psp` file containing the HTML code. Name the procedure `show_cust`.

   b. Use a parameter to pass `CUSTOMER_ID` and then display information for that customer.

   c. Use an HTML form to call the PSP. Modify the `lab_05_02b.psp` file and add the necessary details to call the PSP.

Fine-Grained Access Control

Objectives

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

- **Describe the process of fine-grained access control**
- **Implement and test fine-grained access control**

Objectives

In this lesson, you will learn about the security features in the Oracle Database from an application developer's standpoint.

For more information about these features, refer to *Oracle Supplied PL/SQL Packages and Types Reference, Oracle Label Security Administrator's Guide, Oracle Single Sign-On Application Developer's Guide,* and *Oracle Security Overview*.

Overview

Fine-grained access control:

- **Enables you to enforce security through a low level of granularity**
- **Restricts users to viewing only "their" information**
- **Is implemented through a security policy attached to tables**
- **Dynamically modifies user statements to fit the policy**
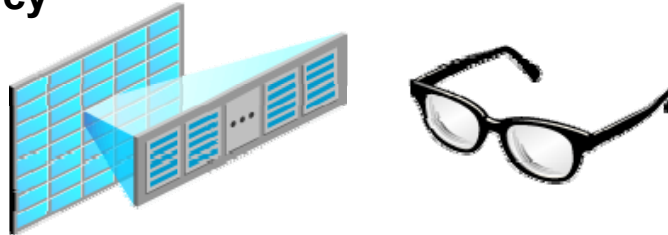
Fine-Grained Access Control

Fine-grained access control enables you to build applications that enforce security rules (or policies) at a low level of granularity. For example, you can use fine-grained access to restrict customers who access the Oracle server to see only their own account, physicians to see only the records of their own patients, or managers to see only the records of employees who work for them.

When you use fine-grained access control, you create security policy functions attached to the table or view on which you have based your application. Then, when a user enters a DML statement on that object, the Oracle server dynamically modifies the user's statement—transparently to the user—so that the statement implements the correct access control.

Fine-grained access is also known as a virtual private database (VPD) because it implements row-level security, essentially giving the user access to his or her own private database. Fine-grained means at the individual row level.

Identifying Fine-Grained Access Features
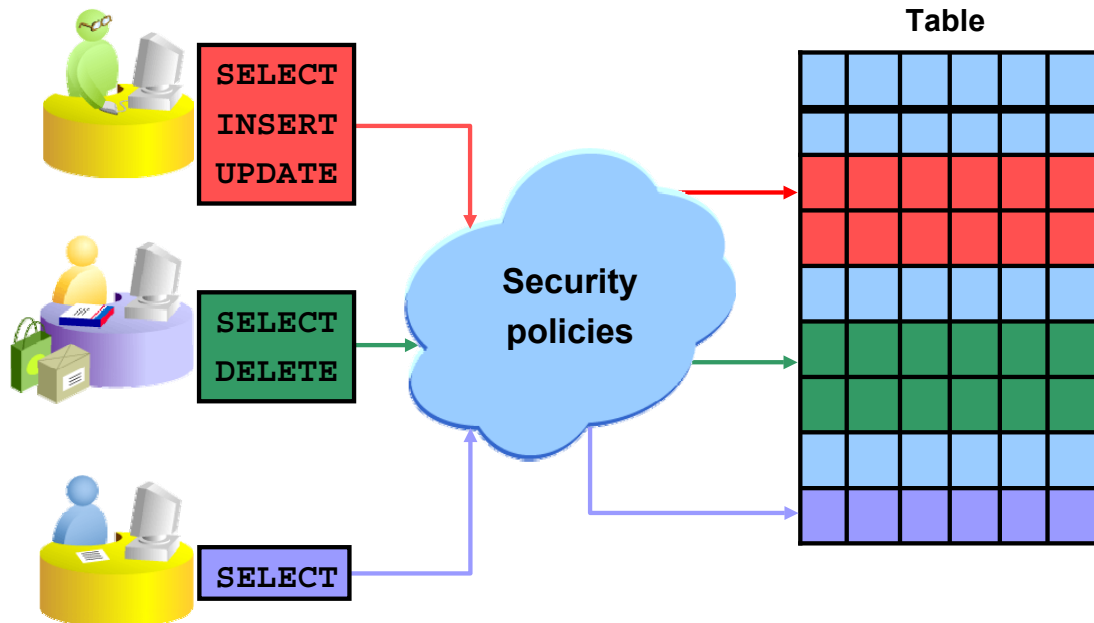
Features

You can use fine-grained access control to implement security rules called policies with functions, and then associate those security policies with tables or views. The database server automatically enforces those security policies, no matter how the data is accessed.

A security policy is a collection of rules needed to enforce the appropriate privacy and security rules into the database itself, making it transparent to users of the data structure.

Attaching security policies to tables or views, rather than to applications, provides greater security, simplicity, and flexibility.

You can:
- Use different policies for `SELECT`, `INSERT`, `UPDATE`, and `DELETE` statements
- Use security policies only where you need them
- Use more than one policy for each table, including building on top of base policies in packaged applications
- Distinguish policies between different applications by using policy groups

How Fine-Grained Access Works

Implement the policy on the `CUSTOMERS` **table:**
"Account managers can see only their own
customers."
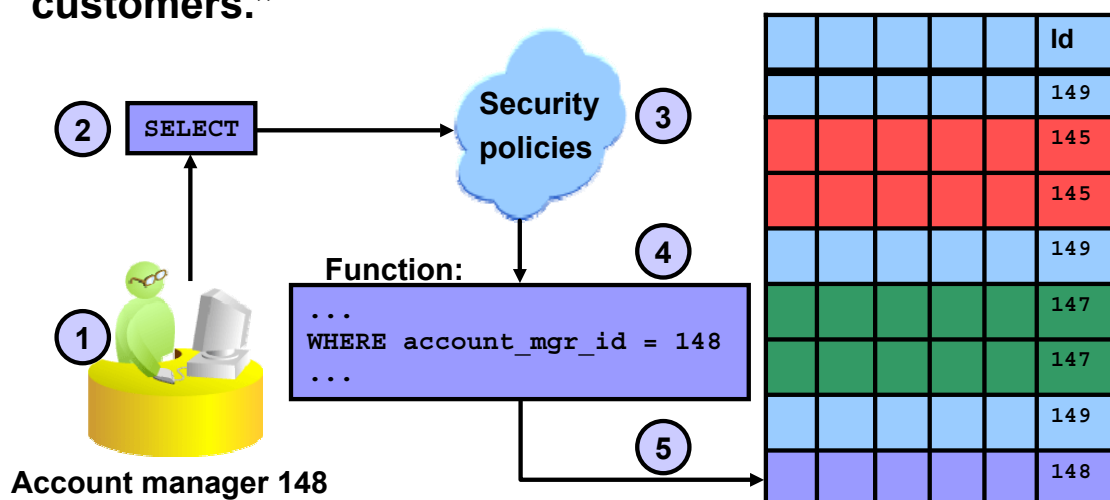
Account manager 148

How Fine-Grained Access Works

To implement a virtual private database so that each account manager can see only his or her own customers, you must do the following:

1. Create a function to add a `WHERE` clause identifying a selection criterion to a user's DML statement.
2. Have the user (the account manager) enter a DML statement.
3. Implement the security policy through the function you created. The Oracle server calls the function automatically.
4. Dynamically modify the user's statement through the function.
5. Execute the dynamically modified statement.

How Fine-Grained Access Works

- ## You write a function to return the account manager ID:

```
account_mgr_id = (SELECT account_mgr_id
                  FROM   customers
                  WHERE  account_mgr_id =
                  SYS_CONTEXT ('userenv','session_user'));
```

- ## The account manager user enters a query:

```
SELECT customer_id, cust_last_name, cust_email
FROM   customers;
```

- ## The query is modified with the function results:

```
SELECT customer_id, cust_last_name, cust_email
FROM   orders
WHERE  account_mgr_id = (SELECT account_mgr_id
                  FROM   customers
                  WHERE  account_mgr_id =
                  SYS_CONTEXT ('userenv','session_user'));
```

ORACLE

How Fine-Grained Access Works (continued)

Fine-grained access control is based on a dynamically modified statement. In the example shown, the user enters a broad query against the CUSTOMERS table that retrieves customer names and e-mail names for a specific account manager. The Oracle server calls the function to implement the security policy. This modification is transparent to the user. It results in successfully restricting access to other customers' information, displaying only the information relevant to the account manager.

Note: SYS_CONTEXT is a function that returns a value for an attribute. This is explained in detail in a few pages.

Why Use Fine-Grained Access?

To implement the business rule "Account managers can see only their own customers," you have three options:

Option	Comment
Modify all existing application code to include a predicate (a `WHERE` clause) for all SQL statements.	Does not ensure privacy enforcement outside the application. Also, all application code may need to be modified in the future as business rules change.
Create views with the necessary predicates and then create synonyms with the same name as the table names for these views.	This can be difficult to administer, especially if there are a large number of views to track and manage.
Create a VPD for each of the account managers by creating policy functions to generate dynamic predicates. These predicates can then be applied across all objects.	This option offers the best security without major administrative overhead and it also ensures the complete privacy of information

ORACLE

Why Choose Fine-Grained Access?

You can implement the business rule "Account managers can see only their own customers" through a few means. The options are listed above. By using fine-grained access, you have security implemented without a lot of overhead.

Using an Application Context

- **An application context is used to facilitate the implementation of fine-grained access control.**
- **It is a named set of attribute/value pairs associated with a PL/SQL package.**
- **Applications can have their own application-specific context.**
- **Users cannot change their context.**
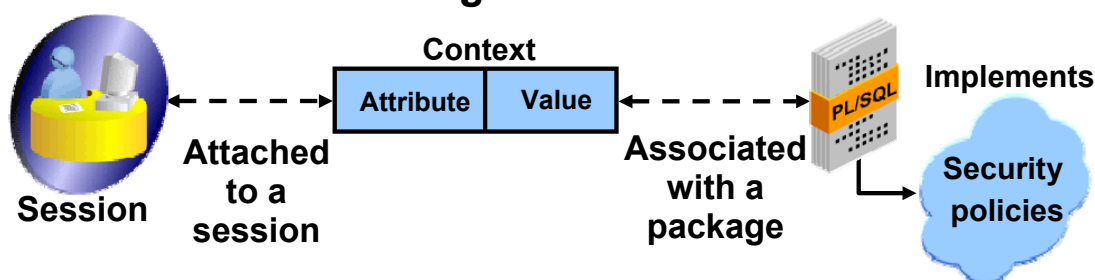
What Is an Application Context?

An application context:
- Is a named set of attribute/value pairs associated with a PL/SQL package
- Is attached to a session
- Enables you to implement security policies with functions and then associate them with applications

A context is a named set of attribute/value pairs that are global to your session. You can define an application context, name it, and associate a value to that context with a PL/SQL package. Application context enables you to write applications that draw upon certain aspects of a user's session information. It provides a way to define, set, and access attributes that an application can use to enforce access control—specifically, fine-grained access control.

Most applications contain information about the basis on which access is to be limited. In an order entry application, for example, you would limit the customers' to access their own orders (ORDER_ID) and customer number (CUSTOMER_ID). Or, you may limit an account manager (ACCOUNT_MGR_ID) to view only his or her customers. These values can be used as security attributes. Your application can use a context to set values that are accessed within your code and used to generate WHERE clause predicates for fine-grained access control.

An application context is owned by SYS.

Using an Application Context

System defined:

USERENV Context

Attribute	Value
IP_ADDRESS	139.185.35.118
SESSION_USER	oe
CURRENT_SCHEMA	oe
DB_NAME	orcl

Application defined:

YOUR_DEFINED Context

Attribute	Value
customer_info	cus_1000
account_mgr	AM145

The function SYS_CONTEXT returns a value of an attribute of a context.

```
SELECT SYS_CONTEXT ('USERENV', 'SESSION_USER')
FROM DUAL;

SYS_CONTEXT ('USERENV', 'SESSION_USER')
------------------------------------------------------
 OE
```

Application Context

A predefined application context named USERENV is available to you. It has a predefined list of attributes. Predefined attributes can be very useful for access control. You find the values of the attributes in a context by using the SYS_CONTEXT function. Although the predefined attributes in the USERENV application context are accessed with the SYS_CONTEXT function, you cannot change them.

With the SYS_CONTEXT function, you pass the context name and the attribute name. The attribute value is returned.

The following statement returns the name of the database being accessed:

```
SELECT SYS_CONTEXT ('USERENV', 'DB_NAME')
FROM DUAL;

SYS_CONTEXT('USERENV','DB_NAME')
-----------------------------------------------
ORCL
```

Creating an Application Context

```
CREATE [OR REPLACE] CONTEXT namespace
USING   [schema.]plsql_package
```

- **Requires the CREATE ANY CONTEXT system privilege**
- **Parameters:**
  - **namespace is the name of the context.**
  - **schema is the name of the schema owning the PL/SQL package.**
  - **plsql_package is the name of the package used to set or modify the attributes of the context. (It does not need to exist at the time of the context creation.)**

```
CREATE CONTEXT order_ctx USING oe.orders_app_pkg;

Context created.
```

Creating a Context

For fine-grained access where you want an account manager to view only his or her customers, customers can view their own information, and sales representatives can view only their own orders, you can create a context called ORDER_CTX and define for it the ACCOUNT_MGR, CUST_ID, and SALE_REP attributes.

Because a context is associated with a PL/SQL package, you need to name the package that you are tying to the context. This package does not need to exist at the time of context creation.

Setting a Context

- **Use the supplied package procedure DBMS_SESSION.SET_CONTEXT to set a value for an attribute within a context.**

```
DBMS_SESSION.SET_CONTEXT('context_name',
                         'attribute_name',
                         'attribute_value')
```

- **Set the attribute value in the package associated to the context.**

```
CREATE OR REPLACE PACKAGE orders_app_pkg
...
BEGIN
    DBMS_SESSION.SET_CONTEXT('ORDER_CTX',
                             'ACCOUNT_MGR',
                              v_user)
...
```

Setting a Context

When a context is defined, you can use the DBMS_SESSION.SET_CONTEXT procedure to set a value for an attribute within a context. The attribute is set in the package associated with the context.

```
CREATE OR REPLACE PACKAGE orders_app_pkg
IS
 PROCEDURE set_app_context;
END;
/
CREATE OR REPLACE PACKAGE BODY orders_app_pkg
IS
 c_context CONSTANT VARCHAR2(30) := 'ORDER_CTX';
 PROCEDURE set_app_context
 IS
    v_user VARCHAR2(30);
 BEGIN
  SELECT user INTO v_user FROM dual;
  DBMS_SESSION.SET_CONTEXT
    (c_context, 'ACCOUNT_MGR', v_user);
 END;
END;
/
```

Setting a Context (continued)

In the example on the previous page, the context ORDER_CTX has the ACCOUNT_MGR attribute set to the current user logged (determined by the USER function).

For this example, assume that users AM145, AM147, AM148, and AM149 exist. As each user logs on and the DBMS_SESSION.SET_CONTEXT is invoked, the attribute value for that ACCOUNT_MGR is set to the user ID.

```
GRANT EXECUTE ON oe.orders_app_pkg
  TO AM145, AM147, AM148, AM149;

CONNECT AM145/oracle
Connected.

EXECUTE oe.orders_app_pkg.set_app_context

SELECT SYS_CONTEXT('ORDER_CTX', 'ACCOUNT_MGR') FROM dual;

SYS_CONTEXT('ORDER_CTX', 'ACCOUNT_MGR')
-----------------------------------------------------------
AM145
```

If you switch the user ID, the attribute value is also changed to reflect the current user.

```
CONNECT AM147/oracle
Connected.

EXECUTE oe.orders_app_pkg.set_app_context

SELECT SYS_CONTEXT('ORDER_CTX', 'ACCOUNT_MGR') FROM dual;

SYS_CONTEXT('ORDER_CTX', 'ACCOUNT_MGR')
-----------------------------------------------------------
AM147
```

Implementing a Policy

Follow these steps:

1. Set up a driving context.

```
CREATE OR REPLACE CONTEXT order_ctx
  USING orders_app_pkg;
```

2. Create the package associated with the context you defined in step 1. In the package:

   a. Set the context.

   b. Define the predicate.

3. Define the policy.

4. Set up a logon trigger to call the package at logon time and set the context.

5. Test the policy.

ORACLE

Implementing a Policy

In this example, assume that the users `AM145`, `AM147`, `AM148`, and `AM149` exist. Next, a context and a package associated with the context is created. The package will be owned by `OE`.

Step 1: Set Up a Driving Context

Use the `CREATE CONTEXT` syntax to create a context.

```
CONNECT /AS sysdba

CREATE CONTEXT order_ctx USING oe.orders_app_pkg;
```

Oracle University and En-Sof Informatica E Treinamento Ltda use only

Step 2: Creating the Package

```
CREATE OR REPLACE PACKAGE orders_app_pkg
IS
 PROCEDURE show_app_context;
 PROCEDURE set_app_context;
 FUNCTION the_predicate
   (p_schema VARCHAR2, p_name VARCHAR2)
    RETURN VARCHAR2;
END orders_app_pkg;    -- package spec
/
```

Implementing a Policy (continued)

Step 2: Create a Package

In the OE schema, the ORDERS_APP_PKG is created. This package contains three routines:

- **show_app_context:** For learning and testing purposes, this procedure will display a context attribute and value.
- **set_app_context:** This procedure sets a context attribute to a specific value.
- **the_predicate:** This function builds the predicate (the WHERE clause) that will control the rows visible in the CUSTOMERS table to a user. (Note that this function requires two input parameters. An error will occur when the policy is implemented if you exclude these two parameters.)

Implementing a Policy (continued)

Step 2: Create a Package (continued)

```
CREATE OR REPLACE PACKAGE BODY orders_app_pkg
IS
  c_context CONSTANT VARCHAR2(30) := 'ORDER_CTX';
  c_attrib  CONSTANT VARCHAR2(30) := 'ACCOUNT_MGR';

PROCEDURE show_app_context
IS
BEGIN
  DBMS_OUTPUT.PUT_LINE('Type: ' || c_attrib ||
   ' - ' || SYS_CONTEXT(c_context, c_attrib));
END show_app_context;

PROCEDURE set_app_context
  IS
    v_user VARCHAR2(30);
BEGIN
  SELECT user INTO v_user FROM dual;
  DBMS_SESSION.SET_CONTEXT
    (c_context, c_attrib, v_user);
END set_app_context;

FUNCTION the_predicate
(p_schema VARCHAR2, p_name VARCHAR2)
RETURN VARCHAR2
IS
  v_context_value VARCHAR2(100) :=
    SYS_CONTEXT(c_context, c_attrib);
  v_restriction VARCHAR2(2000);
BEGIN
  IF v_context_value LIKE 'AM%'  THEN
    v_restriction :=
     'ACCOUNT_MGR_ID =
      SUBSTR(''' || v_context_value || ''', 3, 3)';
  ELSE
    v_restriction := null;
  END IF;
  RETURN v_restriction;
END the_predicate;

END orders_app_pkg; -- package body
/
```

Note that the THE_PREDICATE function builds the WHERE clause and stores it in the
V_RESTRICTION variable. If the SYS_CONTEXT function returns an attribute value that starts
with AM, then the WHERE clause is built with ACCOUNT_MGR_ID = *the last three characters of
the attribute value*. If the user is AM145, then the WHERE clause will be:

```
WHERE account_mgr_id = 145
```

Step 3: Defining the Policy

Use the `DBMS_RLS` package:

- **It contains the fine-grained access administrative interface.**
- **It adds a fine-grained access control policy to a table or view.**
- **You use the `ADD_POLICY` procedure to add a fine-grained access control policy to a table or view.**

`ADD_POLICY` ⟶ **`oe_access_policy`**

Security policies
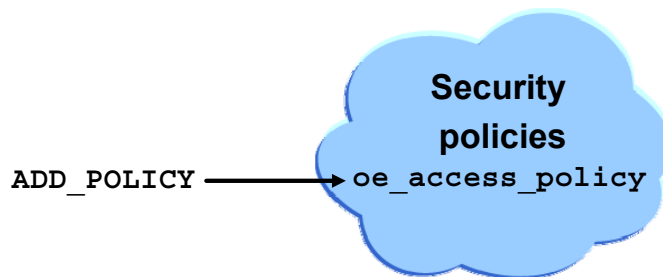
Implementing a Policy (continued)

The `DBMS_RLS` package contains the fine-grained access control administrative interface. The package holds several procedures. To add a policy, you use the `ADD_POLICY` procedure within the `DBMS_RLS` package.

Note: `DBMS_RLS` is available with the Enterprise Edition only.

Step 3: Define the Policy

The `DBMS_RLS.ADD_POLICY` procedure adds a fine-grained access control policy to a table or view. The procedure causes the current transaction, if any, to commit before the operation is carried out. However, this does not cause a commit first if it is inside a DDL event trigger. These are the parameters for the `ADD_POLICY` procedure:

```
DBMS_RLS.ADD_POLICY (
   object_schema    IN VARCHAR2 := NULL,
   object_name      IN VARCHAR2,
   policy_name      IN VARCHAR2,
   function_schema  IN VARCHAR2 := NULL,
   policy_function  IN VARCHAR2,
   statement_types  IN VARCHAR2 := NULL,
   update_check     IN BOOLEAN := FALSE,
   enable           IN BOOLEAN := TRUE);
```

Implementing a Policy (continued)

Step 3: Define the Policy (continued)

Parameter	Description
OBJECT_SCHEMA	Schema containing the table or view (logon user, if NULL)
OBJECT_NAME	Name of table or view to which the policy is added
POLICY_NAME	Name of policy to be added. It must be unique for the same table or view.
FUNCTION_SCHEMA	Schema of the policy function (logon user, if NULL)
POLICY_FUNCTION	Name of a function that generates a predicate for the policy. If the function is defined within a package, then the name of the package must be present.
STATEMENT_TYPES	Statement types that the policy will apply. It can be any combination of SELECT, INSERT, UPDATE, and DELETE. The default is to apply to all these types.
UPDATE_CHECK	Optional argument for the INSERT or UPDATE statement types. The default is FALSE. Setting update_check to TRUE causes the server to also check the policy against the value after insert or update.
ENABLE	Indicates if the policy is enabled when it is added. The default is TRUE.

Below is a list of the procedures contained in the DBMS_RLS package. For detailed information, refer to the *PL/SQL Packages and Types Reference 10g Release 1 (10.1)* reference manual.

Procedure	Description
ADD_POLICY	Adds a fine-grained access control policy to a table or view
DROP_POLICY	Drops a fine-grained access control policy from a table or view
REFRESH_POLICY	Causes all the cached statements associated with the policy to be reparsed
ENABLE_POLICY	Enables or disables a fine-grained access control policy
CREATE_POLICY_GROUP	Creates a policy group
ADD_GROUPED_POLICY	Adds a policy associated with a policy group
ADD_POLICY_CONTEXT	Adds the context for the active application
DELETE_POLICY_GROUP	Deletes a policy group
DROP_GROUPED_POLICY	Drops a policy associated with a policy group
DROP_POLICY_CONTEXT	Drops a driving context from the object so that it will have one less driving context
ENABLE_GROUPED_POLICY	Enables or disables a row-level group security policy
REFRESH_GROUPED_POLICY	Reparses the SQL statements associated with a refreshed policy

Step 3: Defining the Policy

```
CONNECT /as sysdba

DECLARE
BEGIN
  DBMS_RLS.ADD_POLICY (
    'OE',                            ⟵  Object schema
    'CUSTOMERS',                     ⟵  Table name
    'OE_ACCESS_POLICY',              ⟵  Policy name
    'OE',                            ⟵  Function schema
    'ORDERS_APP_PKG.THE_PREDICATE',  ⟵  Policy function
    'SELECT, UPDATE, DELETE',        ⟵  Statement types
    FALSE,                           ⟵  Update check
    TRUE);                           ⟵  Enabled
END;
/
```

Implementing a Policy (continued)

Step 3: Define the Policy (continued)

The security policy OE_ACCESS_POLICY is created and added with the
DBMS_RLS.ADD_POLICY procedure. The predicate function that defines how the policy is to
be implemented is associated with the policy being added.

This example specifies that whenever a SELECT, UPDATE, or DELETE statement on the
OE.CUSTOMERS table is executed, the predicate function return result is appended to the end of
the WHERE clause.

Step 4: Setting Up a Logon Trigger

Create a database trigger that executes whenever anyone logs on to the database:

```
CONNECT /as sysdba


CREATE OR REPLACE TRIGGER set_id_on_logon
AFTER logon on DATABASE
BEGIN
  oe.orders_app_pkg.set_app_context;
END;
/
```

Implementing a Policy (continued)

Step 4: Set Up a Logon Trigger

After the context is created, the security package is defined, the predicate is defined, and the policy is defined, you need to create a logon trigger to implement fine-grained access control. This trigger causes the context to be set as each user is logged on.

Viewing Example Results

Data in the `CUSTOMERS` table:

```
CONNECT oe/oe
SELECT   COUNT(*), account_mgr_id
FROM     customers
GROUP BY account_mgr_id;

  COUNT(*) ACCOUNT_MGR_ID
---------- --------------
       111 145
        76 147
        58 148
        74 149
         1
```

```
CONNECT AM148/oracle
SELECT   customer_id, cust_last_name
FROM     oe.customers;

CUSTOMER_ID CUST_LAST_NAME
----------- ----------------
...
58 rows selected.
```

ORACLE

Example Results

The `AM148` user who logs on will see only the rows in the `CUSTOMERS` table that are defined by the predicate function. The user can issue `SELECT`, `UPDATE` and `DELETE` statements against the `CUSTOMERS` table, but only the rows defined by the predicate function can be manipulated.

```
UPDATE oe.customers
SET credit_limit = credit_limit + 5000
WHERE customer_id = 101;

0 rows updated.
```

The `AM148` user does not have access to customer ID 101. Customer ID 101 has the account manager of 145. To user `AM148`, any updates, deletes, or selects attempted on customers that do not have him as an account manager are not performed. It is as though those customers do not exist.

Using Data Dictionary Views

- **USER_POLICIES**
- **ALL_POLICIES**
- **DBA_POLICIES**
- **ALL_CONTEXT**
- **DBA_CONTEXT**

Data Dictionary Views

You can query the data dictionary views to find out information about the policies available in your schema.

View	Description
USER_POLICIES	All policies owned by the current schema
ALL_POLICIES	All policies owned or accessible by the current schema
DBA_POLICIES	All policies
ALL_CONTEXT	All active context namespaces defined in the session
DBA_CONTEXT	All context namespace information (active and inactive)

Using the `ALL_CONTEXT` Dictionary View

Use `ALL_CONTEXT` to see the active context namespaces defined in your session:

```
CONNECT AM148/oracle

SELECT *
FROM   all_context;

NAMESPACE             SCHEMA       PACKAGE
----------------      ---------    -----------
ORDER_CTX             OE           ORDERS_APP_PKG
```

Dictionary Views

You can use the `ALL_CONTEXT` dictionary view to view information about contexts to which you have access. In the slide, the `NAMESPACE` column is equivalent to the context name.

You can use the `ALL_POLICIES` dictionary view to view information about polices to which you have access. In the example below, information is shown on the `OE_ACCESS_POLICY` policy.

```
SELECT object_name, policy_name, pf_owner, package,
       function, sel, ins, upd, del
FROM all_policies;

OBJECT_NAME                   POLICY_NAME
----------------------------  ----------------------------
PF_OWNER                      PACKAGE
----------------------------  ----------------------------
FUNCTION                      SEL INS UPD DEL
----------------------------  --- --- --- ---
CUSTOMERS                     OE_ACCESS_POLICY
OE                            ORDERS_APP_PKG
THE_PREDICATE                 YES NO  YES YES
```

Policy Groups

- **Indicate a set of policies that belong to an application**
- **Are set up by a DBA through an application context, called a driving context**
- **Use the `DBMS_RLS` package to administer the security policies**
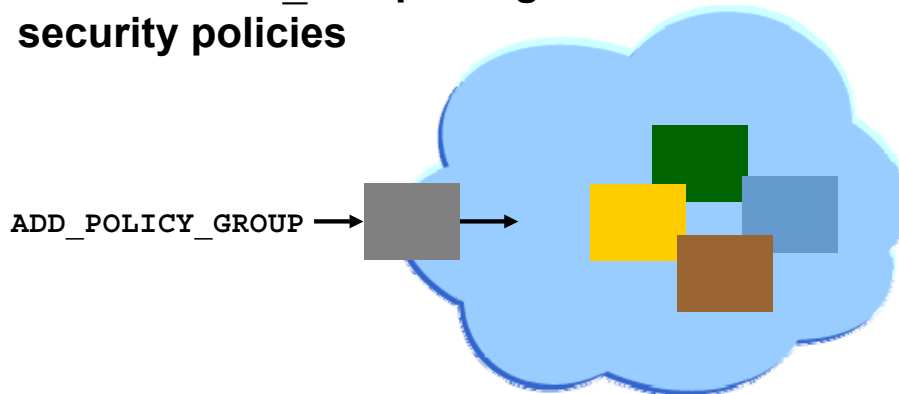
`ADD_POLICY_GROUP` →

ORACLE

Policy Groups

Policy groups were introduced in Oracle9*i*, release 1 (9.0.1). The database administrator designates an application context, called a driving context, to indicate the policy group in effect. When tables or views are accessed, the fine-grained access control engine looks up the driving context to determine the policy group in effect and enforces all the associated policies that belong to that policy group.

The PL/SQL `DBMS_RLS` package enables you to administer your security policies and groups. Using this package, you can add, drop, enable, disable, and refresh the policy groups you create.

More About Policies

- **`SYS_DEFAULT` is the default policy group:**
  - **`SYS_DEFAULT` group may or may not contain policies.**
  - **All policies belong to `SYS_DEFAULT` by default.**
  - **You cannot drop the `SYS_DEFAULT` policy group.**
- **Use `DBMS_RLS.CREATE_POLICY_GROUP` to create a new group.**
- **Use `DBMS_RLS.ADD_GROUPED_POLICY` to add a policy associated with a policy group.**
- **You can apply multiple driving contexts to the same table or view.**

ORACLE

More About Policies

A policy group is a set of security policies that belong to an application. You can designate an application context (known as a driving context) to indicate the policy group in effect. When the tables or views are accessed, the server looks up the driving context (that is also known as policy context) to determine the policy group in effect. It enforces all the associated policies that belong to that policy group.

By default, all policies belong to the SYS_DEFAULT policy group. Policies defined in this group for a particular table or view will always be executed along with the policy group specified by the driving context. The SYS_DEFAULT policy group may or may not contain policies. If you attempt to drop the SYS_DEFAULT policy group, an error will be raised. If you add policies associated with two or more objects to the SYS_DEFAULT policy group, then each such object will have a separate SYS_DEFAULT policy group associated with it. For example, the CUSTOMERS table in the OE schema has one SYS_DEFAULT policy group, and the ORDERS table in the OE schema has a different SYS_DEFAULT policy group associated with it. If you add policies associated with two or more objects, then each such object will have a separate SYS_DEFAULT policy group associated with it.

```
SYS_DEFAULT
 - policy1 (OE/CUSTOMERS)
 - policy3 (OE/CUSTOMERS)
SYS_DEFAULT
 - policy2 (OE/ORDERS)
```

THESE eKIT MATERIALS ARE FOR YOUR USE IN THIS CLASSROOM ONLY. COPYING eKIT MATERIALS FROM THIS
COMPUTER IS STRICTLY PROHIBITED

Oracle Database 10g: Advanced PL/SQL   6-24

Oracle University and En-Sof Informatica E Treinamento Ltda use only

More About Policies (continued)

When adding the policy to a table or view, you can use the
`DBMS_RLS.ADD_GROUPED_POLICY` interface to specify the group to which the policy
belongs. To specify which policies will be effective, you add a driving context using the
`DBMS_RLS.ADD_POLICY_CONTEXT` interface. If the driving context returns an unknown
policy group, an error is returned.

If the driving context is not defined, then all policies are executed. Likewise, if the driving
context is `NULL`, then policies from all policy groups are enforced. In this way, an application
that accesses the data cannot bypass the security setup module (that sets up application context)
to avoid any applicable policies.

You can apply multiple driving contexts to the same table or view, and each of them will be
processed individually. In this way, you can configure multiple active sets of policies to be
enforced.

You can create a new policy using the `DBMS_RLS` package either from the command line or
programmatically, or access the Oracle Policy Manager graphical user interface in Oracle
Enterprise Manager.

Summary

In this lesson, you should have learned how to:

- **Describe the process of fine-grained access control**
- **Implement and test fine-grained access control**
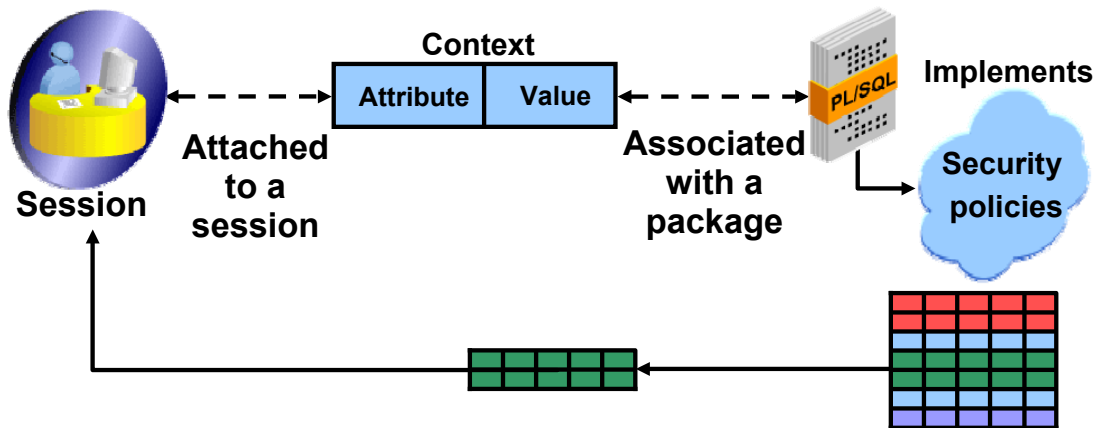
Summary

In this lesson you should have learned about fine-grained access control and the steps required to implement a virtual private database.

Practice Overview

This practice covers the following topics:

- **Creating an application context**
- **Creating a policy**
- **Creating a logon trigger**
- **Implementing a virtual private database**
- **Testing the virtual private database**

ORACLE

Practice Overview

In this practice you will implement and test fine-grained access control.

Practice 6

In this practice you will define an application context and security policy to implement the policy: "Sales Representatives can see their own order information only in the ORDERS table." You will create sales representative IDs to test the success of your implementation.

Examine the definition of the ORDERS table, and the sales representative's data:

```
DESCRIBE orders
Name                   Null?     Type
------------------ -------- ---------------------------------
ORDER_ID               NOT NULL NUMBER(12)
ORDER_DATE             NOT NULL TIMESTAMP(6) WITH LOCAL TIME ZONE
ORDER_MODE                      VARCHAR2(8)
CUSTOMER_ID            NOT NULL NUMBER(6)
ORDER_STATUS                    NUMBER(2)
ORDER_TOTAL                     NUMBER(8,2)
SALES_REP_ID                    NUMBER(6)
PROMOTION_ID                    NUMBER(6)

SELECT sales_rep_id, count(*)
FROM   orders
GROUP BY sales_rep_id;

SALES_REP_ID   COUNT(*)
------------ ----------
         153          5
         154         10
         155          5
         156          5
         158          7
         159          7
         160          6
         161         13
         163         12
                     35
10 rows selected.
```

1. Examine, and then run the lab_06_01.sql script.
   This script will create the sales representative's ID accounts with appropriate privileges to access the database.

2. Set up an application context:

   Connect to the database as SYSDBA before creating this context.

   Create an application context named sales_orders_ctx.

   Associate this context to the oe.sales_orders_pkg.

Practice 6 (continued)

3. Connect as OE/OE.

   Examine this package specification:
   ```
   CREATE OR REPLACE PACKAGE sales_orders_pkg
   IS
    PROCEDURE set_app_context;
    FUNCTION the_predicate
      (p_schema VARCHAR2, p_name VARCHAR2)
       RETURN VARCHAR2;
   END sales_orders_pkg;    -- package spec
   /
   ```

   Create this package specification and then the package body in the OE schema.

   When you create the package body, set up two constants as follows:
   ```
   c_context CONSTANT VARCHAR2(30) := 'SALES_ORDER_CTX';
   c_attrib  CONSTANT VARCHAR2(30) := 'SALES_REP';
   ```

Use these constants in the SET_APP_CONTEXT procedure to set the application context to the current user.

4. Connect as SYSDBA and define the policy.

   Use DBMS_RLS.ADD_POLICY to define the policy.

   Use these specifications for the parameter values:
   ```
   object_schema    OE
   object_name      ORDERS
   policy_name      OE_ORDERS_ACCESS_POLICY
   function_schema  OE
   policy_function  SALES_ORDERS_PKG.THE_PREDICATE
   statement_types  SELECT, INSERT, UPDATE, DELETE
   update_check     FALSE,
   enable           TRUE);
   ```

5. Connect as SYSDBA and create a logon trigger to implement fine-grained access control. You can call the trigger SET_ID_ON_LOGON. This trigger causes the context to be set as each user is logged on.

Practice 6 (continued)

6. Test the fine-grained access implementation. Connect as your `SR` user and query the `ORDERS` table. For example, your results should match:

```
CONNECT sr153/oracle

SELECT sales_rep_id, COUNT(*)
FROM   orders
GROUP BY sales_rep_id;

SALES_REP_ID   COUNT(*)
------------ ----------
         153          5

CONNECT sr154/oracle

SELECT sales_rep_id, COUNT(*)
FROM   orders
GROUP BY sales_rep_id;

SALES_REP_ID   COUNT(*)
------------ ----------
         154         10
```

Note

During debugging, you may need to disable or remove some of the objects created for this lesson.

If you need to disable the logon trigger, issue the command:
```
ALTER TRIGGER set_id_on_logon DISABLE;
```

If you need to remove the policy you created, issue the command:
```
EXECUTE DBMS_RLS.DROP_POLICY('OE', 'ORDERS', -
    'OE_ORDERS_ACCESS_POLICY')
```

Performance and Tuning

Objectives

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

- **Tune PL/SQL code**
- **Identify and tune memory issues**
- **Recognize network issues**
- **Perform native and interpreted compilation**

Objectives

In this lesson, the performance and tuning topics are divided into four main groups

- Tuning PL/SQL code
- Memory issues
- Network issues
- Native and interpreted compilation.

In the "Tuning PL/SQL Code" section, you learn why it is important to write smaller executable sections of code; when to use SQL or PL/SQL; how bulk binds can improve performance; how to use the FORALL syntax; how to rephrase conditional statements; about data types and constraint issues.

In the memory issues section, you learn about the shared pool and what you can do programmatically to tune it.

In the network issues section, you learn why it is important to group your OPEN-FOR statements when passing host cursor variables to PL/SQL; when it is appropriate to use client-side PL/SQL; how to avoid unnecessary reparsing; how to utilize array processing.

In the compilation section, you learn about native and interpreted compilation.

Tuning PL/SQL Code

You can tune your PL/SQL code by:

- **Writing smaller executable sections of code**
- **Comparing SQL with PL/SQL and where one is appropriate over the other**
- **Understanding how bulk binds can improve performance**
- **Using the `FORALL` support with bulk binding**
- **Handling and saving exceptions with the `SAVE EXCEPTIONS` syntax**
- **Rephrasing conditional statements**
- **Identifying data type and constraint issues**

ORACLE

Tuning PL/SQL Code

By tuning your PL/SQL code, you can tailor its performance to best meet your needs. In the following pages you learn about some of the main PL/SQL tuning issues that can improve the performance of your PL/SQL applications.

Modularizing Your Code

- **Limit the number of lines of code between a `BEGIN` and `END` to about a page or 60 lines of code.**
- **Use packaged programs to keep each executable section small.**
- **Use local procedures and functions to hide logic.**
- **Use a function interface to hide formulas and business rules.**

Write Smaller Executable Sections

By writing smaller sections of executable code, you can make the code easier to read, understand, and maintain. When developing an application, use a stepwise refinement. Make a general description of what you want your program to do, and then implement the details in subroutines. Using local modules and packaged programs can help in keeping each executable section small. This will make it easier for you to debug and refine your code.

Comparing SQL with PL/SQL

Each has its own benefits:

- **SQL:**
  - **Accesses data in the database**
  - **Treats data as sets**
- **PL/SQL:**
  - **Provides procedural capabilities**
  - **Has more flexibility built into the language**

SQL Versus PL/SQL

Both SQL and PL/SQL have their strengths. However, there are situations where one language is more appropriate to use than the other.

You use SQL to access data in the database with its powerful statements. SQL processes sets of data as groups rather than as individual units. The flow-control statements of most programming languages are absent in SQL, but present in PL/SQL. When using SQL in your PL/SQL applications, be sure not to repeat a SQL statement. Instead, encapsulate your SQL statements in a package and make calls to the package.

Using PL/SQL, you can take advantage of the PL/SQL-specific enhancements for SQL, such as autonomous transactions, fetching into cursor records, using a cursor `FOR` loop, using the `RETURNING` clause for information about modified rows, and using `BULK COLLECT` to improve the performance of multirow queries.

While there are advantages of using PL/SQL over SQL in several cases, use PL/SQL with caution, especially under the following circumstances:
- Performing high-volume inserts
- Using user-defined PL/SQL functions
- Using external procedure calls
- Using the `utl_file` package as an alternative to SQL*Plus in high-volume reporting

Comparing SQL with PL/SQL

- **Some simple set processing is markedly faster than the equivalent PL/SQL.**

```
BEGIN
  INSERT INTO inventories2
    SELECT product_id, warehouse_id
    FROM main_inventories;
END;
```

- **Avoid using procedural code when it may be better to use SQL.**

```
...FOR I IN 1..5600 LOOP
    counter := counter + 1;
    SELECT product_id, warehouse_id
      INTO v_p_id, v_wh_id
      FROM big_inventories WHERE v_p_id = counter;
    INSERT INTO inventories2 VALUES(v_p_id, v_wh_id);
  END LOOP;...
```

ORACLE

Copyright © 2004, Oracle. All rights reserved.

SQL Versus PL/SQL (continued)

The SQL statement explained in the slide is a great deal faster than the equivalent PL/SQL loop.
Take advantage of the simple set processing operations implicitly available in the SQL language,
as it can run markedly faster than the equivalent PL/SQL loop. Avoid writing procedural code
when SQL would work better.

However, there are occasions when you will get better performance from PL/SQL even when the
process could be written in SQL. Correlated updates are slow. With correlated updates, a better
method is to access only correct rows using PL/SQL. The following PL/SQL loop is faster than
the equivalent correlated update SQL statement.

```
DECLARE
  CURSOR cv_raise IS
    SELECT deptno, increase
    FROM emp_raise;
BEGIN
    FOR dept IN cv_raise LOOP
      UPDATE big_emp
        SET sal = sal * dept.increase
        WHERE deptno = dept.deptno;
    END LOOP;
  ...
```

THESE eKIT MATERIALS ARE FOR YOUR USE IN THIS CLASSROOM ONLY. COPYING eKIT MATERIALS FROM THIS
COMPUTER IS STRICTLY PROHIBITED

Comparing SQL with PL/SQL

- **Instead of:**

```
...
INSERT INTO order_items
   (order_id, line_item_id, product_id,
    unit_price, quantity)
VALUES (...
```

- **Create a stand-alone procedure:**

```
insert_order_item (
   2458, 6, 3515, 2.00, 4);
```

- **Or, a packaged procedure:**

```
orderitems.ins (
   2458, 6, 3515, 2.00, 4);
```

ORACLE

Encapsulate SQL Statements

From a design standpoint, do not embed your SQL statements directly within application code. It is better if you write procedures to perform your SQL statements.

Pros
- If you design your application so that all programs that perform an insert on a specific table use the same INSERT statement, your application will run faster because of less parsing and reduced demands on the SGA memory.
- Your program will also handle DML errors consistently.

Cons
- You may need to write more procedural code.
- You may need to write several variations of update or insert procedures to handle the combinations of columns that you are updating or inserting into.

Using Bulk Binding

Use bulk binds to reduce context switches between the PL/SQL engine and the SQL engine.

```
                  PL/SQL run-time engine    →        SQL engine
                                            ←

  PL/SQL block

  FORALL j IN 1..1000        Procedural          SQL
    INSERT …                 statement           statement
    (OrderId(j),             executor            executor
    OrderDate(j), …);
```
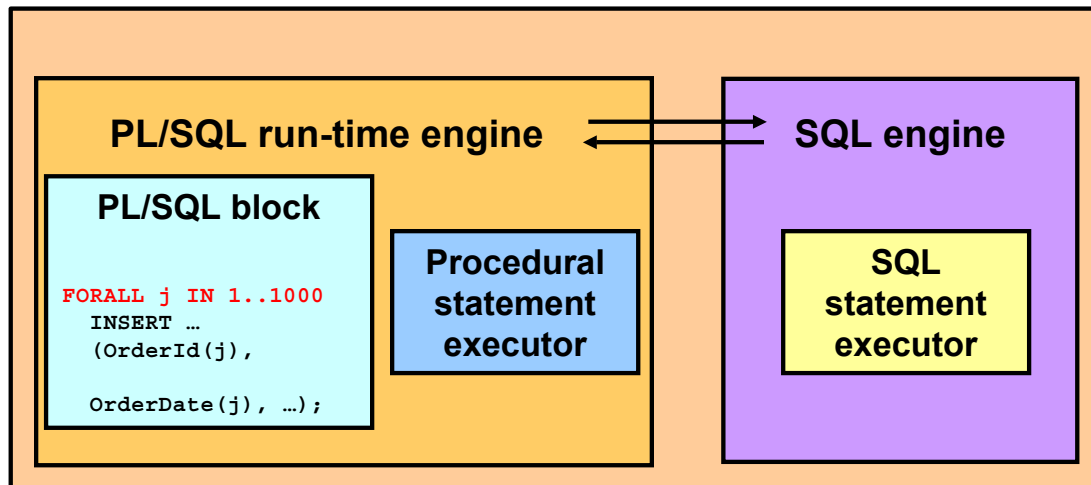
Bulk Binding

With bulk binds, you can improve performance by decreasing the number of context switches between the SQL and PL/SQL engine. When a PL/SQL program executes, each time a SQL statement is encountered, there is a switch between the PL/SQL engine to the SQL engine. The more the number of switches, the lesser the efficiency.

Improved Performance

Bulk binding enables you to implement array fetching. With bulk binding, entire collections, and not just individual elements, are passed back and forth. Bulk binding can be used with nested tables, varrays, and associative arrays.

The more rows affected by a SQL statement, the greater is the performance gain with bulk binding.

Using Bulk Binding

Bind whole arrays of values all at once, rather than looping to perform fetch, insert, update, and delete on multiple rows.

- **Instead of:**

```
...
FOR i IN 1 .. 50000 LOOP
  INSERT INTO bulk_bind_example_tbl
    VALUES(...);
END LOOP; ...
```

- **Use:**

```
...
FORALL i IN 1 .. 50000
  INSERT INTO bulk_bind_example_tbl
    VALUES(...);
END; ...
```

ORACLE

Using Bulk Binding

In the first example shown, one row is inserted into the target table at a time. In the second example, the FOR loop is changed to a FORALL (which has an implicit loop) and all immediately subsequent DML statements are processed in bulk. The following are the entire code examples along with timing statistics for running each FOR loop example.

First, create the demonstration table:

```
CREATE TABLE bulk_bind_example_tbl (
  num_col NUMBER,
  date_col DATE,
  char_col VARCHAR2(40));
```

Second, set the SQL*Plus TIMING variable on. Setting this on enables you to see the approximate elapsed time of the last SQL statement:

```
SET TIMING ON
```

Third, run this block of code that includes a FOR loop to insert 50,000 rows:

```
DECLARE
  TYPE typ_numlist IS TABLE OF NUMBER;
  TYPE typ_datelist IS TABLE OF DATE;
  TYPE typ_charlist IS TABLE OF VARCHAR2(40)
    INDEX BY PLS_INTEGER;
  -- continued onto next page...
```

Using Bulk Binding (continued)

```
      n typ_numlist  := typ_numlist();
      d typ_datelist := typ_datelist();
      c typ_charlist;

   BEGIN
     FOR i IN 1 .. 50000 LOOP
       n.extend;
       n(i) := i;
       d.extend;
       d(i) := sysdate + 1;
       c(i) := lpad(1, 40);
     END LOOP;
     FOR I in 1 .. 50000 LOOP
       INSERT INTO bulk_bind_example_tbl
         VALUES (n(i), d(i), c(i));
     END LOOP;
   END;
   /
   PL/SQL procedure successfully completed.
   Elapsed: 00:00:17.62
```

Last, run this block of code that includes a FORALL loop to insert 50,000 rows. Note the significant decrease in the timing when using the FORALL processing:

```
   DECLARE
     TYPE typ_numlist IS TABLE OF NUMBER;
     TYPE typ_datelist IS TABLE OF DATE;
     TYPE typ_charlist IS TABLE OF VARCHAR2(40)
       INDEX BY PLS_INTEGER;

     n typ_numlist  := typ_numlist();
     d typ_datelist := typ_datelist();
     c typ_charlist;

   BEGIN
     FOR i IN 1 .. 50000 LOOP
       n.extend;
       n(i) := i;
       d.extend;
       d(i) := sysdate + 1;
       c(i) := lpad(1, 40);
     END LOOP;
     FORALL I in 1 .. 50000
       INSERT INTO bulk_bind_example_tbl
         VALUES (n(i), d(i), c(i));
   END;
   /

   PL/SQL procedure successfully completed.
   Elapsed: 00:00:02.08
```

Using Bulk Binding

Use `BULK COLLECT` to improve performance:

```
CREATE OR REPLACE PROCEDURE process_customers
  (p_account_mgr customers.account_mgr_id%TYPE)
IS
  TYPE typ_numtab IS TABLE OF
    customers.customer_id%TYPE;
  TYPE typ_chartab IS TABLE OF
    customers.cust_last_name%TYPE;
  TYPE typ_emailtab IS TABLE OF
    customers.cust_email%TYPE;
  v_custnos     typ_numtab;
  v_last_names typ_chartab;
  v_emails      typ_emailtab;
BEGIN
  SELECT customer_id, cust_last_name, cust_email
    BULK COLLECT INTO v_custnos, v_last_names, v_emails
    FROM customers
    WHERE account_mgr_id = p_account_mgr;
  ...
END process_customers;
```

Oracle University and En-Sof Informatica E Treinamento Ltda use only

Using `BULK COLLECT`

When you require a large number of rows to be returned from the database, you can use the `BULK COLLECT` option for queries. This option enables you to retrieve multiple rows of data in a single request. The retrieved data is then populated into a series of collection variables. This query will run significantly faster than if it were done without the `BULK COLLECT`.

You can use the `BULK COLLECT` option with explicit cursors too:

```
BEGIN
  OPEN cv_customers INTO customers_rec;
  FETCH cv_customers BULK COLLECT INTO
    v_custnos, v_last_name, v_mails;
  ...
```

Using Bulk Binding

Use the `RETURNING` clause to retrieve information about rows being modified:

```
DECLARE
  TYPE      typ_replist IS VARRAY(100) OF NUMBER;
  TYPE      typ_numlist IS TABLE OF
              orders.order_total%TYPE;
  repids    typ_replist :=
              typ_replist(153, 155, 156, 161);
  totlist   typ_numlist;
  c_big_total CONSTANT NUMBER := 60000;
BEGIN
  FORALL i IN repids.FIRST..repids.LAST
    UPDATE   orders
    SET      order_total = .95 * order_total
    WHERE    sales_rep_id = repids(i)
    AND      order_total > c_big_total
    RETURNING order_total BULK COLLECT INTO Totlist;
  END;
```

Oracle University and En-Sof Informatica E Treinamento Ltda use only

The `RETURNING` Clause

Often, applications need information about the row affected by a SQL operation, for example, to generate a report or take a subsequent action. Using the `RETURNING` clause, you can retrieve information about rows you have just modified with the `INSERT`, `UPDATE`, and `DELETE` statements. This can improve performance because it enables you to make changes, and at the same time, collect information of the data being changed. As a result, fewer network round trips, less server CPU time, fewer cursors, and less server memory are required. Without the `RETURNING` clause, you need two operations: one to make the change, and a second operation to retrieve information about the change.

In the example shown, the `order_total` information is retrieved from the `ORDERS` table and collected into the `totlist` collection. The `totlist` collection is returned in bulk to the PL/SQL engine.

If you did not use the `RETURNING` clause, you would need to perform two operations, one for the `UPDATE`, and another for the `SELECT`:

```
        UPDATE  orders SET order_total = .95 * order_total
        WHERE   sales_rep_id = p_id
        AND     order_total > c_big_total;

        SELECT order_total FROM   orders
        WHERE  sales_rep_id = p_id AND order_total > c_big_total;
```

The RETURNING Clause (continued)

In the following example, you update the credit limit of a customer and at the same time retrieve the customer's new credit limit into a SQL*Plus environment variable:

```
CREATE OR REPLACE PROCEDURE change_credit
    (p_in_id   IN   customers.customer_id%TYPE,
     o_credit OUT NUMBER)
    IS
    BEGIN
     UPDATE customers
     SET    credit_limit = credit_limit * 1.10
     WHERE  customer_id = p_in_id
     RETURNING credit_limit INTO o_credit;
END change_credit;
/
VARIABLE g_credit NUMBER
EXECUTE change_credit(109, :g_credit)
PRINT g_credit
```

Using SAVE EXCEPTIONS

- **You can use the SAVE EXCEPTIONS keywords in your FORALL statements:**

```
FORALL index IN lower_bound..upper_bound
SAVE EXCEPTIONS
{insert_stmt | update_stmt | delete_stmt}
```

- **Exceptions raised during execution are saved in the %BULK_EXCEPTIONS cursor attribute.**

- **The attribute is a collection of records with two fields:**

Field	Definition
ERROR_INDEX	Holds the iteration of the FORALL statement where the exception was raised
ERROR_CODE	Holds the corresponding Oracle error code

- – **Note that the values always refer to the most recently executed FORALL statement.**

ORACLE

Handling FORALL Exceptions

To handle exceptions encountered during a BULK BIND operation, you can add the keyword SAVE EXCEPTIONS to your FORALL statement. Without it, if any one row fails during the FORALL loop, the loop execution is terminated. SAVE_EXCEPTIONS allows the loop to continue processing and is required if you want the loop to continue.

All exceptions raised during the execution are saved in the cursor attribute %BULK_EXCEPTIONS, which stores a collection of records. This cursor attribute is available only from the exception handler.

Each record has two fields. The first field, %BULK_EXCEPTIONS(i).ERROR_INDEX, holds the "iteration" of the FORALL statement during which the exception was raised. The second field, BULK_EXCEPTIONS(i).ERROR_CODE, holds the corresponding Oracle error code.

The values stored by %BULK_EXCEPTIONS always refer to the most recently executed FORALL statement. The number of exceptions is saved in the count attribute of %BULK_EXCEPTIONS, that is, %BULK_EXCEPTIONS.COUNT. Its subscripts range from 1 to COUNT. If you omit the keywords SAVE EXCEPTIONS, execution of the FORALL statement stops when an exception is raised. In that case, SQL%BULK_EXCEPTIONS.COUNT returns 1, and SQL%BULK_EXCEPTIONS contains just one record. If no exception is raised during the execution, SQL%BULK_EXCEPTIONS.COUNT returns 0.

Handling `FORALL` Exceptions

```
DECLARE
  TYPE NumList IS TABLE OF NUMBER;
  num_tab    NumList :=
         NumList(100,0,110,300,0,199,200,0,400);
  bulk_errors EXCEPTION;
  PRAGMA        EXCEPTION_INIT (bulk_errors, -24381 );
BEGIN
  FORALL i IN num_tab.FIRST..num_tab.LAST
  SAVE EXCEPTIONS
  DELETE FROM orders WHERE order_total < 500000/num_tab(i);
EXCEPTION WHEN bulk_errors THEN
  DBMS_OUTPUT.PUT_LINE('Number of errors is: '
                       || SQL%BULK_EXCEPTIONS.COUNT);
  FOR j in 1..SQL%BULK_EXCEPTIONS.COUNT
  LOOP
    DBMS_OUTPUT.PUT_LINE (
      TO_CHAR(SQL%BULK_EXCEPTIONS(j).error_index) ||
      ' / ' ||
      SQLERRM(-SQL%BULK_EXCEPTIONS(j).error_code) );
  END LOOP;
END;
/
```

Example

In this example, the EXCEPTION_INIT pragma defines an exception named BULK_ERRORS
and associates the name to the ORA-24381 code, which is an "Error in Array DML".
The PL/SQL block raises the predefined exception ZERO_DIVIDE when i equals 2, 5, 8. After
the bulk-bind is completed, SQL%BULK_EXCEPTIONS.COUNT returns 3 because of trying to
divide by zero three times. To get the Oracle error message (which includes the code), we pass
SQL%BULK_EXCEPTIONS(i).ERROR_CODE to the error-reporting function SQLERRM.
Here is the output:

```
        Number of errors is: 5
        Number of errors is: 3
        2 / ORA-01476: divisor is equal to zero
        5 / ORA-01476: divisor is equal to zero
        8 / ORA-01476: divisor is equal to zero

        PL/SQL procedure successfully completed.
```
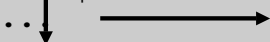
Rephrasing Conditional Control Statements

In logical expressions, PL/SQL stops evaluating the expression as soon as the result is determined.

- **Scenario 1:**

```
IF TRUE|FALSE OR (v_sales_rep_id IS NULL) THEN
    ..
    ...
END IF;
```

- **Scenario 2:**

```
IF credit_ok(cust_id) AND (v_order_total < 5000)  THEN
   ...
END IF;
```

Rephrase Conditional Control Statements

In logical expressions, improve performance by tuning conditional constructs carefully.

When evaluating a logical expression, PL/SQL stops evaluating the expression as soon as the result can be determined. For example, in the first scenario in the slide, which involves an OR expression, when the value of the left operand yields TRUE, PL/SQL need not evaluate the right operand (because OR returns TRUE if either of its operands is true).

Now, consider the second scenario in the slide, which involves an AND expression. The Boolean function CREDIT_OK is always called. However, if you switch the operands of AND as follows, the function is called only when the expression v_order_total < 5000 is true (because AND returns TRUE only if both its operands are true):

```
        IF (v_order_total < 5000 ) AND credit_ok(cust_id) THEN
               ...
        END IF;
```

Rephrasing Conditional Control Statements

If your business logic results in one condition being true, use the `ELSIF` syntax for mutually exclusive clauses:

```
IF v_acct_mgr = 145 THEN
  process_acct_145;
END IF;
IF v_acct_mgr = 147 THEN
  process_acct_147;
END IF;
IF v_acct_mgr = 148 THEN
  process_acct_148;
END IF;
IF v_acct_mgr = 149 THEN
  process_acct_149;
END IF;
```

```
IF v_acct_mgr = 145
THEN
  process_acct_145;
ELSIF v_acct_mgr = 147
THEN
  process_acct_147;
ELSIF v_acct_mgr = 148
THEN
  process_acct_148;
ELSIF v_acct_mgr = 149
THEN
  process_acct_149;
END IF;
```

ORACLE

Oracle University and En-Sof Informatica E Treinamento Ltda use only

Mutually Exclusive Conditions

If you have a situation where you are checking a list of choices for a mutually exclusive result, use the `ELSIF` syntax, as it offers the most efficient implementation. With `ELSIF`, after a branch evaluates to TRUE, the other branches are not executed.

In the example shown on the right, every `IF` statement is executed. In the example on the left, after a branch is found to be true, the rest of the branch conditions are not evaluated.

Sometimes you do not need an `IF` statement. For example, the following code can be rewritten without an `IF` statement:

```
IF date_ordered < sysdate + 7 THEN
  late_order := TRUE;
ELSE
  late_order := FALSE;
END IF;

--rewritten without an IF statement:
late_order := date_ordered < sysdate + 7;
```

Avoiding Implicit Data Type Conversion

- **PL/SQL performs implicit conversions between structurally different data types.**
- **Example: When assigning a `PLS_INTEGER` variable to a `NUMBER` variable**

```
DECLARE
  n NUMBER;
BEGIN
  n := n + 15;     -- converted
  n := n + 15.0;   -- not converted
  ...
END;
```

numbers   **dates**   **strings**

Avoid Implicit Data Type Conversion

PL/SQL automatically performs implicit conversions between structurally different types at run time for you. By avoiding implicit conversions, you can improve the performance of your code. The major problems with implicit data type conversion are:

- It is non-intuitive and can result in unexpected results.
- You have no control over the implicit conversion.

In the example shown, assigning a `PLS_INTEGER` variable to a `NUMBER` variable or vice versa results in a conversion, because their representations are different. Such implicit conversions can happen during parameter passing as well. The integer literal 15 is represented internally as a signed 4-byte quantity, so PL/SQL must convert it to an Oracle number before the addition. However, the floating-point literal 15.0 is represented as a 22-byte Oracle number, so no conversion is necessary.

To avoid implicit data type conversion, you can use the built-in functions:

- `TO_DATE`
- `TO_NUMBER`
- `TO_CHAR`
- `CAST`

Using `PLS_INTEGER` Data Type for Integers

Use `PLS_INTEGER` when dealing with integer data:

- **It is an efficient data type for integer variables.**
- **It requires less storage than `INTEGER` or `NUMBER`.**
- **Its operations use machine arithmetic, which is faster than library arithmetic.**

Use `PLS_INTEGER` for All Integer Operations

When you need to declare an integer variable, use the `PLS_INTEGER` data type, which is the most efficient numeric type. That is because `PLS_INTEGER` values require less storage than `INTEGER` or `NUMBER` values, which are represented internally as 22-byte Oracle numbers. Also, `PLS_INTEGER` operations use machine arithmetic, so they are faster than `BINARY_INTEGER`, `INTEGER`, or `NUMBER` operations, which use library arithmetic.

Furthermore, `INTEGER`, `NATURAL`, `NATURALN`, `POSITIVE`, `POSITIVEN`, and `SIGNTYPE` are constrained subtypes. Their variables require precision checking at run time that can affect the performance.

The Oracle Database 10*g* data types `BINARY_FLOAT` and `BINARY_DOUBLE` are also faster than the `NUMBER` data type.

Understanding the `NOT NULL` Constraint

```
PROCEDURE calc_m IS
 m NUMBER NOT NULL:=0;
 a NUMBER;
 b NUMBER;
BEGIN
 ...
 m := a + b;
 ...
END;
```

```
PROCEDURE calc_m IS
m NUMBER; --no
          --constraint
a NUMBER;
b NUMBER;
BEGIN
  ...
 m := a + b;
 IF m IS NULL THEN
   -- raise error
 END IF;
END;
```

The `NOT NULL` Constraint

In PL/SQL, using the `NOT NULL` constraint incurs a small performance cost. Therefore, use it with care. Consider the example on the left in the slide that uses the `NOT NULL` constraint for *m*.

Because *m* is constrained by `NOT NULL`, the value of the expression *a* + *b* is assigned to a temporary variable, which is then tested for nullity. If the variable is not null, its value is assigned to *m*. Otherwise, an exception is raised. However, if *m* were not constrained, the value would be assigned to *m* directly.

A more efficient way to write the same example is shown on the right in the slide.

Note that the subtypes `NATURALN` and `POSTIVEN` are defined as `NOT NULL` subtypes of `NATURAL` and `POSITIVE`. Using them incurs the same performance cost as seen above.

Using the `NOT NULL` Constraint	Not Using the Constraint
Slower	Faster
No extra coding is needed.	Requires extra coding that is error prone
When an error is implicitly raised, the value of *m* is preserved.	When an error is explicitly raised, the old value of *m* is lost.

Passing Data Between PL/SQL Programs

- **The flexibility built into PL/SQL enables you to pass:**
  - **Simple scalar variables**
  - **Complex data structures**
- **You can use the `NOCOPY` hint to improve performance with `IN OUT` parameters.**
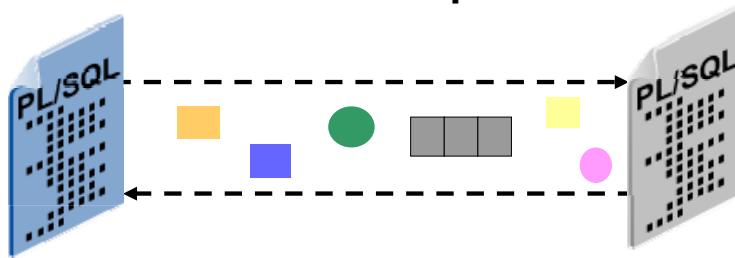
Passing Data Between PL/SQL Programs

You can pass simple scalar data or complex data structures between PL/SQL programs.

When passing collections as parameters, you may encounter a slight decrease in performance as compared with passing scalar data, but the performance is still comparable. However, when passing `IN OUT` parameters that are complex (such as collections) to a procedure, you will experience significantly more overhead because a copy of the parameter value before the routine is executed is stored. The stored value must be kept in case an exception occurs. You can use the `NOCOPY` compiler hint to improve performance in this situation. `NOCOPY` instructs the compiler to not make a backup copy of the parameter that is being passed. Be careful when using the `NOCOPY` compiler hint because should your program encounter an exception, your results are not predictable.

Passing Data Between PL/SQL Programs

Passing records as parameters to encapsulate data, as well as, write and maintain less code:

```
DECLARE
  TYPE CustRec IS RECORD (
    customer_id      customers.customer_id%TYPE,
    cust_last_name   VARCHAR2(20),
    cust_email       VARCHAR2(30),
    credit_limit     NUMBER(9,2));
  ...
  PROCEDURE raise_credit (cust_info CustRec);
```

ORACLE

Passing Records as Arguments

You can declare user-defined records as formal parameters of procedures and functions as shown above. By using records to pass values, you are encapsulating the data being passed, and it requires less coding than defining, assigning, and manipulating each record field individually.

When you call a function that returns a record, use the notation:

```
function_name(parameters).field_name
```

For example, the following call to the NTH_HIGHEST_ORD_TOTAL function references the field ORDER_TOTAL in the ORD_INFO record:

```
DECLARE
  TYPE OrdRec IS RECORD (
  v_order_id     NUMBER(6),
  v_order_total  REAL);
  v_middle_total REAL;
FUNCTION nth_highest_total (n INTEGER) RETURN OrdRec IS
  order_info OrdRec;
BEGIN  ...
  RETURN order_info;  -- return record
END;
BEGIN                 -- call function
  v_middle_total := nth_highest_total(10).v_order_total;
```

Passing Data Between PL/SQL Programs

Use collections as arguments:

```
PACKAGE cust_actions IS
  TYPE NameTabTyp IS TABLE OF
customer.cust_last_name%TYPE
    INDEX BY PLS_INTEGER;
  TYPE CreditTabTyp IS TABLE OF
customers.credit_limit%TYPE
    INDEX BY PLS_INTEGER;
  ...
  PROCEDURE credit_batch( name_tab   IN NameTabTyp ,
                          credit_tab IN CreditTabTyp,
                          ...);
  PROCEDURE log_names ( name_tab IN NameTabTyp );
END cust_actions;
```

Passing Collections as Arguments

You can declare collections as formal parameters of procedures and functions. In the example in the slide, associative arrays are declared as the formal parameters of two packaged procedures. If you were to use scalar variables to pass the data, you would need to code and maintain many more declarations.

Identifying and Tuning Memory Issues

Tuning the shared pool:
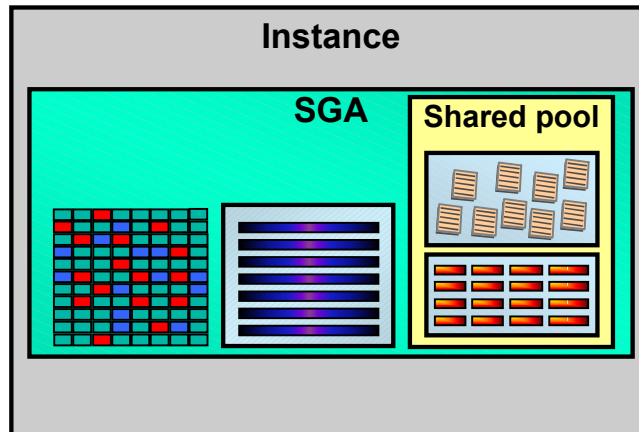
Tuning the Size of the Shared Pool of the SGA

When you invoke a program element, such as a procedure or a package, its compiled version is loaded into the shared pool memory area, if it is not already present there. It remains there until the memory is needed by other resources and the package has not been used recently. If it gets flushed out from memory, the next time any object in the package is needed, the whole package has to be loaded in memory again, which involves time and maintenance to make space for it.

If the package is already present in the shared memory area, your code executes faster. It is, therefore, important to make sure that packages that are used very frequently are always present in memory. The larger the shared pool area, the more likely it is that the package remains in memory. However, if the shared pool area is too large, you waste memory. When tuning the shared pool, make sure it is large enough to hold all the frequently needed objects in your application.

Note: Tuning the shared pool is usually a DBA's responsibility.

Pinning Objects

Pinning:

- **Is used so that objects avoid the Oracle least recently used (LRU) mechanism and do not get flushed out of memory**
- **Is applied with the help of the `sys.dbms_shared_pool` package:**
  - `sys.dbms_shared_pool.keep`
  - `sys.dbms_shared_pool.unkeep`
  - `sys.dbms_shared_pool.sizes`
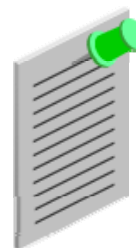
ORACLE

What Is Pinning a Package?

Sizing the shared pool properly is one of the ways of ensuring that frequently used objects are available in memory whenever needed, so that performance improves. Another way to improve performance is to pin frequently used packages in the shared pool.

When a package is pinned, it is not aged out with the normal least recently used (LRU) mechanism that the Oracle server otherwise uses to flush out a least recently used package. The package remains in memory no matter how full the shared pool gets or how frequently you access the package.

You pin packages with the help of the sys.dbms_shared_pool package. This package contains three procedures:

Procedure	Description
keep	Use this procedure to pin objects to the shared pool.
unkeep	Use this procedure to age out an object that you have requested to be kept in the shared pool.
sizes	Use this procedure to dump the contents of the shared pool to the DBMS_OUTPUT buffer. It can show the objects in the shared pool that are larger than the specified size, in kilobytes.

Pinning Objects

Syntax:

```
SYS.DBMS_SHARED_POOL.KEEP(object_name, flag)
```

```
SYS.DBMS_SHARED_POOL.UNKEEP(object_name, flag)
```

Example:

```
...
BEGIN
  SYS.DBMS_SHARED_POOL.KEEP ('OE.OVER_PACK', 'P');
  ...
  SYS.DBMS_SHARED_POOL.UNKEEP ('OE. OVER_PACK', 'P');
  ...
END;
...
```

ORACLE

Using `sys.dbms_shared_pool`

You can pin and unpin packages, procedures, functions, types, triggers, and sequences. This may be useful for certain semi-frequently used large objects (larger than 20 KB), because when large objects are brought into the shared pool, a larger number of other objects (much more than the size of the object being brought in) may need to be aged out in order to create a contiguous area large enough. Pinning occurs when the `sys.dbms_shared_pool.keep` procedure is invoked.

To create DBMS_SHARED_POOL, run the DBMSPOOL.SQL script. The PRVTPOOL.PLB script is automatically executed after DBMSPOOL.SQL runs.

Using `sys.dbms_shared_pool` (continued)

Syntax Definitions

where:		
	object_name	Name of the object to keep
	flag	(Optional) If this is not specified, then the package assumes that the first parameter is the name of a package/procedure/function and resolves the name.
		`'P'` or `'p'` indicates a package/procedure/function. This is the default.
		`'T'` or `'t'` indicates a type.
		`'R'` or `'r'` indicates a trigger.

Pinning Objects

- **Pin objects only when necessary.**
- **The `keep` procedure first queues an object for pinning before loading it.**
- **Pin all objects soon after instance startup to ensure contiguous blocks of memory.**

Guidelines for Pinning Objects

- Pin objects only when necessary. Otherwise, you may end up setting aside too much memory, which can have a negative impact on performance.
- The `keep` procedure does not immediately load a package into the shared pool; it queues the package for pinning. The package is loaded into the shared pool only when the package is first referenced, either to execute a module or to use one of its declared objects, such as a global variable or a cursor.
- Pin all your objects in the shared pool as soon after instance startup as possible, so that contiguous blocks of memory can be set aside for large objects.

Note: You can create a trigger that fires when the database is opened (STARTUP). Using this trigger is a good way to pin packages at the very beginning.

Identifying Network Issues
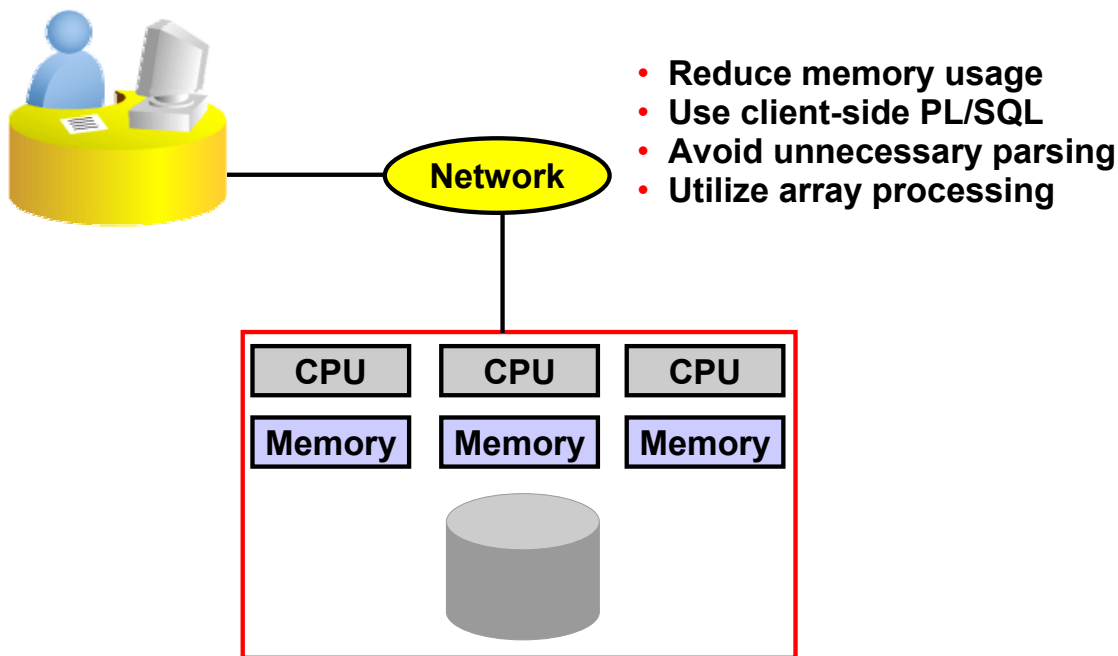
- **Reduce memory usage**
- **Use client-side PL/SQL**
- **Avoid unnecessary parsing**
- **Utilize array processing**

Network

CPU   CPU   CPU

Memory   Memory   Memory

Guidelines for Reducing Network Traffic

Reducing network traffic is one of the key components of tuning because network issues impact performance. When your code is passed to the database, a significant amount of time is spent in the network. The following are some guidelines for reducing network traffic to improve performance:

- When passing host cursor variables to PL/SQL, you can reduce network traffic by grouping `OPEN-FOR` statements. For example, the following PL/SQL block opens five cursor variables in a single round trip:

```
/* anonymous PL/SQL block in host environment */
BEGIN
  OPEN :cust_cv     FOR SELECT * FROM customers;
  OPEN :order_cv    FOR SELECT * FROM orders;
  OPEN :ord_item_cv FOR SELECT * FROM order_items;
  OPEN :wh_cv       FOR SELECT * FROM warehouses;
END;
```

- When you pass host cursor variables to a PL/SQL block for opening, the query work areas to which they point remain accessible after the block completes so that your OCI or Pro*C program can use these work areas for ordinary cursor operations.
- When finished, close the cursors.

Oracle University and En-Sof Informatica E Treinamento Ltda use only

Identifying Network Issues

- **Group `OPEN-FOR` statements when passing host cursor variables to PL/SQL.**
- **Use client-side PL/SQL when appropriate.**
- **Avoid unnecessary reparsing.**
- **Utilize array processing.**
- **Use table functions to improve performance.**
- **Use the `RETURNING` clause when appropriate.**

ORACLE

Guidelines for Reducing Network Traffic (continued)

- If your application is written using development tools that have a PL/SQL engine in the client tool, as in the Oracle Developer tools, and the code is not SQL intensive, reduce the load on the server by doing more of your work in the client and let the client-side PL/SQL engine handle your PL/SQL code.
- When a PL/SQL block is sent from the client to the server, the client can keep a reference to the parsed statement. This reference is the statement handle when using OCI, or the cursor cache entry when using precompilers. If your application is likely to issue the same code more than once, it needs to parse it only the first time. For all subsequent executions, the original parsed statement can be used, possibly with different values for the bind variables. This technique is more appropriate with OCI and precompilers because they give you more control over cursor processing.

  In PL/SQL, this technique can be used with the dbms_sql package, in which the interface is similar to OCI. After a statement is parsed with dbms_sql.parse, it can be executed multiple times.

Guidelines for Reducing Network Traffic (continued)

- OCI and precompilers have the ability to send and retrieve data using host arrays. With this technique, large amounts of data can travel over the network as one unit rather than taking several trips. While PL/SQL does not directly use this array interface, if you are using PL/SQL from OCI or precompilers, take advantage of this interface.
- Use the `RETURNING` clause.
- By using table functions, rows of the result set can be returned a few at a time, reducing the memory overhead for producing large result sets within a function.

Native and Interpreted Compilation

Interpreted code
- **Compiled to m-code**
- **Stored in the database**

PL/SQL source

m-code

Natively compiled code
- **Translated C and compiled**
- **Copied to a code library**

Translated to C code

C compiler

/path/...c

Native code library in OS directory
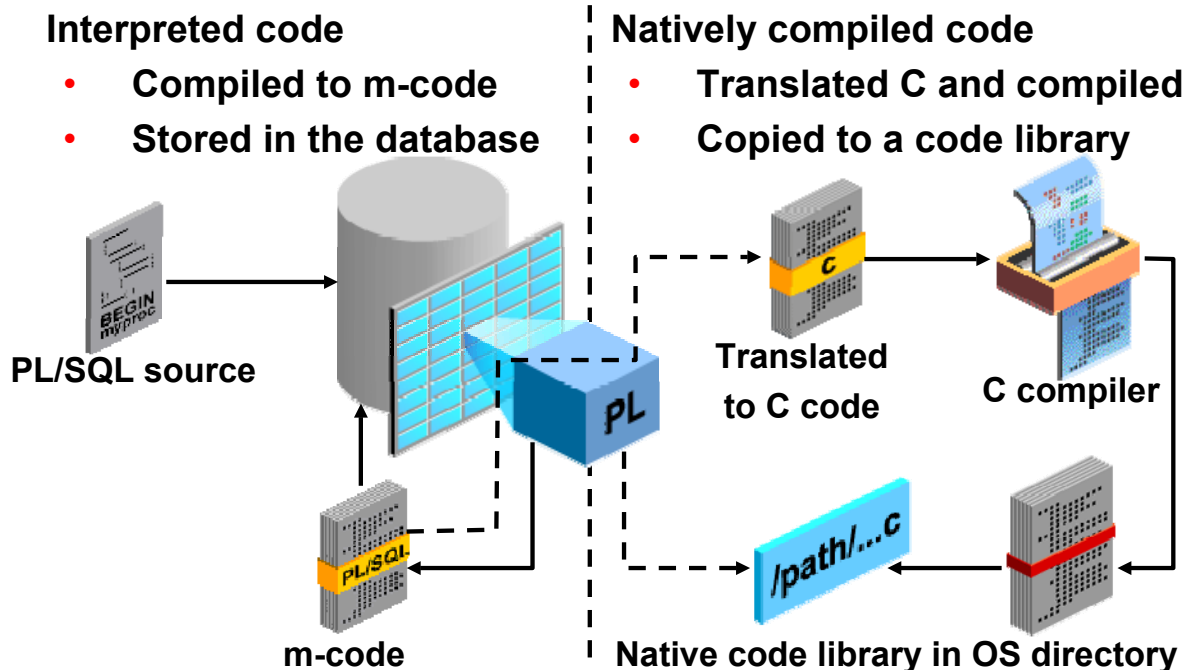
Native and Interpreted Compilation

On the left of the vertical dotted line, a program unit processed as interpreted PL/SQL is compiled into machine-readable code (m-code), which is stored in the database and interpreted at run time.

On the right of the vertical dotted line, the PL/SQL source is subjected to native compilation, where the PL/SQL statements are compiled to m-code that is translated into C code. The m-code is not retained. The C code is compiled with the usual C compiler and linked to the Oracle process using native machine code library. The code library is stored in the database but copied to a specified directory path in the operating system, from which it is loaded at run time. Native code bypasses the typical run-time interpretation of code.

Note: Native compilation cannot do much to speed up SQL statements called from PL/SQL, but it is most effective for computation-intensive PL/SQL procedures that do not spend most of their time executing SQL.

You can natively compile both the supplied Oracle packages and your own PL/SQL code. Compiling all PL/SQL code in the database means that you see the speedup in your own code and all the built-in PL/SQL packages. If you decide that you will have significant performance gains in database operations using PL/SQL native compilation, Oracle recommends that you compile the whole database using the NATIVE setting.

Features and Benefits of Native Compilation

The PL/SQL native compilation process makes use of a makefile, called spnc_makefile.mk, located in the $ORACLE_HOME/plsql directory. The makefile is processed by the Make utility that invokes the C compiler, which is the linker on the supported operating system, to compile and link the resulting C code into shared libraries. The shared libraries are stored inside the database and are copied to the file system. At run time, the shared libraries are loaded and run when the PL/SQL subprogram is invoked.

In accordance with Optimal Flexible Architecture (OFA) recommendations, the shared libraries should be stored near the data files. C code runs faster than PL/SQL, but it takes longer to compile than m-code. PL/SQL native compilation provides the greatest performance gains for computation-intensive procedural operations.

Examples of such operations are data warehouse applications and applications with extensive server-side transformations of data for display. In such cases, expect speed increases of up to 30%.

Limitations of Native Compilation

As stated, the key benefit of natively compiled code is faster execution, particularly for computationally intensive PL/SQL code, as much as 30% more. Consider that:

- Debugging tools for PL/SQL do not handle procedures compiled for native execution. Therefore, use interpreted compilation in development environments, and natively compile the code in a production environment.
- The compilation time increases when using native compilation, because of the requirement to translate the PL/SQL statement to its C equivalent and execute the Make utility to invoke the C compiler and linker for generating the resulting compiled code library.
- If many procedures and packages (more than 5,000) are compiled for native execution, a large number of shared objects in a single directory may affect performance. The operating system directory limitations can be managed by automatically distributing libraries across several subdirectories. To do this, perform the following tasks before natively compiling the PL/SQL code:
    - Set the PLSQL_NATIVE_LIBRARY_SUBDIR_COUNT database initialization parameter to a large value, such as 1,000, before creating the database or compiling the PL/SQL packages or procedures.
    - Create PLSQL_NATIVE_LIBRARY_SUBDIR_COUNT subdirectories in the path specified in the PLSQL_NATIVE_LIBRARY_DIR initialization parameter.

Switching Between Native and Interpreted Compilation

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

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

    - **For the session:**

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

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

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

    - **For the session:**

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

Switching Between Native and Interpreted Compilation

The PLSQL_COMPILER_FLAGS parameter determines whether PL/SQL code is natively compiled or interpreted, and determines whether debug information is included. The default setting is INTERPRETED,NON_DEBUG. To enable PL/SQL native compilation, you must set the value of PLSQL_COMPILER_FLAGS to NATIVE.

If you compile the whole database as NATIVE, then Oracle recommends that you set PLSQL_COMPILER_FLAGS at the system level.

To set compilation type at the system level (usually done by a DBA), execute the following statements:
```
ALTER SYSTEM SET plsql_compiler_flags='NATIVE';
ALTER SYSTEM SET plsql_compiler_flags='INTERPRETED';
```

To set compilation type at the session level, execute one of the following statements:
```
ALTER SESSION SET plsql_compiler_flags='NATIVE';
ALTER SESSION SET plsql_compiler_flags='INTERPRETED';
```

Parameters Influencing Compilation

In all circumstances, whether you intend to compile a database as NATIVE or you intend to compile individual PL/SQL units at the session level, you must set all required parameters.

The system parameters are set in the init*SID*.ora file by using the SPFILE mechanism. Two parameters that are set as system-level parameters are the following:

- The PLSQL_NATIVE_LIBRARY_DIR value, which specifies the full path and directory name used to store the shared libraries that contain natively compiled PL/SQL code
- The PLSQL_NATIVE_LIBRARY_SUBDIR_COUNT value, which specifies the number of subdirectories in the directory specified by the PLSQL_NATIVE_LIBRARY_DIR parameter. Use a script to create directories with consistent names (for example, d0, d1, d2, and so on), and then the libraries are automatically distributed among these subdirectories by the PL/SQL compiler.

By default, PL/SQL program units are kept in one directory.

The PLSQL_COMPILER_FLAGS parameter can be set to a value of NATIVE or INTERPRETED, either as a database initialization for a systemwide default or for each session using an ALTER SESSION statement.

Summary

In this lesson, you should have learned how to:

- **Tune your PL/SQL application. Tuning involves:**
  - **Using the `RETURNING` clause and bulk binds when appropriate**
  - **Rephrasing conditional statements**
  - **Identifying data type and constraint issues**
  - **Understanding when to use SQL and PL/SQL**
- **Tune the shared pool by using the Oracle-supplied package `dbms_shared_pool`**
- **Identify network issues that impact processing**
- **Use native compilation for faster PL/SQL execution**

ORACLE

Summary

There are several methods that help you tune your PL/SQL application.

When tuning PL/SQL code, consider using the `RETURNING` clause and/or bulk binds to improve processing. Be aware of conditional statements with an `OR` clause. Place the fastest processing condition first. There are several data type and constraint issues that can help in tuning an application.

You can use the Oracle-supplied package `dbms_shared_pool` to pin frequently used packages, procedures, and functions to the shared pool.

You can reduce network traffic by:
- Reducing memory usage
- Using client-side PL/SQL
- Avoiding unnecessary parsing
- Utilizing array processing

By using native compilation, you can benefit from performance gains for computation-intensive procedural operations.

Oracle University and En-Sof Informatica E Treinamento Ltda use only

Practice Overview

This practice covers the following topics:

- **Pinning a package**
- **Tuning PL/SQL code to improve performance**
- **Coding with bulk binds to improve performance**

Practice Overview

In this practice, you will tune some of the code you have created for the `OE` application.

- Use `dbms_shared_pool` to pin a package in memory
- Break a previously built subroutine in smaller executable sections
- Pass collections into subroutines
- Add error handling for `BULK INSERT`

For detailed instructions about performing this practice, see Appendix A, "Practice Solutions."

Practice 7

1. In this exercise, you will pin the fine-grained access package created in Lesson 6.
   Note: If you have not completed practice 6, run the following files in the `$HOME/soln` folder:

   ```
   sol_06_02.sql
   sol_06_03.sql
   sol_06_04.sql
   sol_06_05.sql
   ```

   Using the `DBMS_SHARED_POOL.KEEP` procedure, pin your `SALES_ORDERS_PKG`.

   Execute the `DBMS_SHARED_POOL.SIZES` procedure to see the objects in the shared pool that are larger than 500 kilobytes.

2. Open the `lab_07_02.sql` file and examine the package (the package body is shown below):

   ```
   CREATE OR REPLACE PACKAGE BODY credit_card_pkg
   IS

     PROCEDURE update_card_info
       (p_cust_id NUMBER, p_card_type VARCHAR2, p_card_no
      VARCHAR2)
     IS
       v_card_info typ_cr_card_nst;
       i INTEGER;
     BEGIN
       SELECT credit_cards
         INTO v_card_info
         FROM customers
         WHERE customer_id = p_cust_id;
       IF v_card_info.EXISTS(1) THEN   -- cards exist, add more
         i := v_card_info.LAST;
         v_card_info.EXTEND(1);
         v_card_info(i+1) := typ_cr_card(p_card_type,
      p_card_no);
           UPDATE customers
             SET  credit_cards = v_card_info
             WHERE customer_id = p_cust_id;
       ELSE    -- no cards for this customer yet, construct one
         UPDATE customers
           SET  credit_cards = typ_cr_card_nst
                (typ_cr_card(p_card_type, p_card_no))
           WHERE customer_id = p_cust_id;
       END IF;
     END update_card_info;

     -- continued on next page
   ```

Practice 7 (continued)

```
            -- continued from previous page.
              PROCEDURE display_card_info
                (p_cust_id NUMBER)
              IS
                v_card_info typ_cr_card_nst;
                i INTEGER;
              BEGIN
                SELECT credit_cards
                  INTO v_card_info
                  FROM customers
                  WHERE customer_id = p_cust_id;
                IF v_card_info.EXISTS(1) THEN
                  FOR idx IN v_card_info.FIRST..v_card_info.LAST LOOP
                      DBMS_OUTPUT.PUT('Card Type: ' ||
              v_card_info(idx).card_type || ' ');
                      DBMS_OUTPUT.PUT_LINE('/ Card No: ' ||
              v_card_info(idx).card_num );
                  END LOOP;
                ELSE
                  DBMS_OUTPUT.PUT_LINE('Customer has no credit cards.');
                END IF;
              END display_card_info;
            END credit_card_pkg;  -- package body
            /
```

This code needs to be improved. The following issues exist in the code:

- The local variables use the INTEGER data type.
- The same SELECT statement is run in the two procedures.
- The same IF v_card_info.EXISTS(1) THEN statement is in the two procedures.

Practice 7 (continued)

3. To improve the code, make the following modifications:

Change the local `INTEGER` variables to use a more efficient data type.

Move the duplicated code into a function. The package specification for the modification is:

```
CREATE OR REPLACE PACKAGE credit_card_pkg
IS
  FUNCTION cust_card_info
    (p_cust_id NUMBER, p_card_info IN OUT typ_cr_card_nst )
    RETURN BOOLEAN;
  PROCEDURE update_card_info
    (p_cust_id NUMBER, p_card_type VARCHAR2, p_card_no
   VARCHAR2);
   PROCEDURE display_card_info
    (p_cust_id NUMBER);
END credit_card_pkg;  -- package spec
/
```

Have the function return `TRUE` if the customer has credit cards. The function should return `FALSE` if the customer does not have credit cards. Pass into the function an uninitialized nested table. The function places the credit card information into this uninitialized parameter.

4. Test your modified code with the following data:

```
EXECUTE credit_card_pkg.update_card_info –
        (120, 'AM EX', 55555555555)
PL/SQL procedure successfully completed.

EXECUTE credit_card_pkg.display_card_info(120)
Card Type: Visa / Card No: 11111111
Card Type: MC / Card No: 2323232323
Card Type: DC / Card No: 4444444
Card Type: AM EX / Card No: 55555555555

PL/SQL procedure successfully completed.

-- Note: If you did not complete Practice 3, your results
-- will be:

EXECUTE credit_card_pkg.display_card_info(120)
Card Type: AM EX / Card No: 55555555555

PL/SQL procedure successfully completed.
```

Practice 7 (continued)

5. Open file `lab_07_05a.sql`. It contains the modified code from the previous question #3.

    You need to modify the `UPDATE_CARD_INFO` procedure to return information (using the `RETURNING` clause) about the credit cards being updated. Assume that this information will be used by another application developer in your team, who is writing a graphical reporting utility on customer credit cards, after a customer's credit card information is changed.

    Modify the code to use the `RETURNING` clause to find information about the row affected by the `UPDATE` statements.

    You can test your modified code with the following procedure (contained in `lab_07_05b.sql`):

    ```
    CREATE OR REPLACE PROCEDURE test_credit_update_info
    (p_cust_id NUMBER, p_card_type VARCHAR2, p_card_no NUMBER)
    IS
      v_card_info typ_cr_card_nst;
    BEGIN
      credit_card_pkg.update_card_info
        (p_cust_id, p_card_type, p_card_no, v_card_info);
    END test_credit_update_info;
    /
    ```

    Test your code with the following statements set in boldface:

    ```
    EXECUTE test_credit_update_info(125, 'AM EX', 123456789)
    PL/SQL procedure successfully completed.

    SELECT credit_cards FROM customers WHERE customer_id = 125;
    CREDIT_CARDS(CARD_TYPE, CARD_NUM)
    --------------------------------------------------------------
    TYP_CR_CARD_NST(TYP_CR_CARD('AM EX', 123456789))
    ```

Practice 7 (continued)

6. In this exercise, you will test exception handling with the SAVED EXCEPTIONS clause.
   Run the lab_07_06a.sql file to create a test table:

   ```
   CREATE TABLE card_table
   (accepted_cards VARCHAR2(50) NOT NULL);
   ```

   Open the lab_07_06b.sql file and run the contents:

   ```
   DECLARE
     type typ_cards is table of VARCHAR2(50);
     v_cards typ_cards := typ_cards
     ( 'Citigroup Visa', 'Nationscard MasterCard',
       'Federal American Express', 'Citizens Visa',
       'International Discoverer', 'United Diners Club' );
   BEGIN
     v_cards.Delete(3);
     v_cards.DELETE(6);
     FORALL j IN v_cards.first..v_cards.last
       SAVE EXCEPTIONS
       EXECUTE IMMEDIATE
      'insert into card_table (accepted_cards) values (
      :the_card)'
       USING v_cards(j);
   /
   ```

   Note the output:_____

Oracle University and En-Sof Informatica E Treinamento Ltda use only

Practice 7 (continued)

6. (continued)

Open the `lab_07_06c.sql` file and run the contents:

```
 DECLARE
 type typ_cards is table of VARCHAR2(50);
 v_cards typ_cards := typ_cards
 ( 'Citigroup Visa', 'Nationscard MasterCard',
   'Federal American Express', 'Citizens Visa',
   'International Discoverer', 'United Diners Club' );
 bulk_errors EXCEPTION;
 PRAGMA exception_init (bulk_errors, -24381 );
BEGIN
 v_cards.Delete(3);
 v_cards.DELETE(6);
 FORALL j IN v_cards.first..v_cards.last
   SAVE EXCEPTIONS
   EXECUTE IMMEDIATE
  'insert into card_table (accepted_cards) values (
  :the_card)'
   USING v_cards(j);
 EXCEPTION
  WHEN  bulk_errors THEN
    FOR j IN 1..sql%bulk_exceptions.count
  LOOP
    Dbms_Output.Put_Line (
      TO_CHAR( sql%bulk_exceptions(j).error_index ) || ':
      ' || SQLERRM(-sql%bulk_exceptions(j).error_code) );
  END LOOP;
END;
/
```

Note the output:_____

Why is the output different?

Analyzing PL/SQL Code

Oracle University and En-Sof Informatica E Treinamento Ltda use only

Objectives

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

- **Find information about your PL/SQL code**
- **Trace PL/SQL program execution**
- **Profile PL/SQL applications**
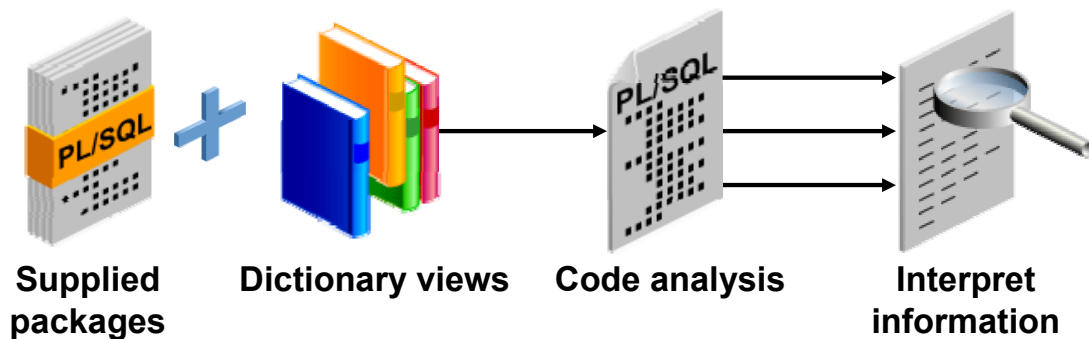
Supplied packages	Dictionary views	Code analysis	Interpret information

Objectives

In this lesson, you learn how to write PL/SQL routines that analyze the PL/SQL applications.
You are introduced to testing PL/SQL code, tracing PL/SQL code, and profiling PL/SQL code.

Finding Coding Information

- **Use the dictionary views:**
  - **ALL_ARGUMENTS**
  - **ALL_OBJECTS**
  - **ALL_SOURCE**
  - **ALL_PROCEDURES**
  - **ALL_DEPENDENCIES**
- **Use the supplied packages:**
  - **dbms_describe**
  - **dbms_utility**

ORACLE

Finding Information on Your PL/SQL Code

The Oracle dictionary views store information on your compiled PL/SQL code. You can write SQL statements against the views to find information about your code.

Dictionary View	Description
ALL_SOURCE	Includes the lines of source code for all the programs you modify
ALL_ARGUMENTS	Includes information about the parameters to the procedures and functions you can call
ALL_PROCEDURES	Contains the list of procedures and functions you can execute
ALL_DEPENDENCIES	Is one of the several views that give you information about dependencies between database objects.

You can also use the Oracle-supplied DBMS_DESCRIBE package to obtain information about a PL/SQL object. The package contains the DESCRIBE_PROCEDURE procedure, which provides a brief description of a PL/SQL stored procedure. It takes the name of a stored procedure and returns information about each parameter of that procedure.

You can use the DBMS_UTILITY supplied package to follow a call stack and an exception stack.

Finding Coding Information

Find all instances of `CHAR` in your code:

```
SELECT NAME, line, text
FROM        user_source
WHERE       INSTR (UPPER(text), ' CHAR') > 0
            OR INSTR (UPPER(text), ' CHAR(') > 0
            OR INSTR (UPPER(text), ' CHAR (') > 0;


NAME               LINE TEXT
---------------- ---- --------------------------------
CUST_ADDRESS_TYP    6    , country_id          CHAR(2)
```

Finding Data Types

You may want to find all occurrences of the `CHAR` data type. The `CHAR` data type is fixed in length and can cause false negatives on comparisons to `VARCHAR2` strings. By finding the `CHAR` data type, you can modify the object, if appropriate, and change it to `VARCHAR2`.

Finding Coding Information

Create a package with various queries that you can easily call:

```
CREATE OR REPLACE PACKAGE query_code_pkg
AUTHID CURRENT_USER
IS
  PROCEDURE find_text_in_code (str IN VARCHAR2);
  PROCEDURE encap_compliance ;
END query_code_pkg;
/
```

Creating a Package to Query Code

A better idea is to create a package to hold various queries that you can easily call. The QUERY_CODE_PKG will hold two validation procedures:

The FIND_TEXT_IN_CODE procedure displays all programs with a specified character string. It queries USER_SOURCE to find occurrences of a text string passed as a parameter. For efficiency, the BULK COLLECT statement is used to retrieve all matching rows into the collection variable.

The ENCAP_COMPLIANCE procedure identifies programs that reference a table directly. This procedure queries the ALL_DEPENDENCIES view to find PL/SQL code objects that directly reference a table or a view.

You can also include a procedure to validate a set of standards for exception handling.

Creating a Package to Query Code (continued)

QUERY_CODE_PKG Code

```
CREATE OR REPLACE PACKAGE BODY query_code_pkg IS
  PROCEDURE find_text_in_code (str IN VARCHAR2)
  IS
    TYPE info_rt IS RECORD (NAME user_source.NAME%TYPE,
      text user_source.text%TYPE );
    TYPE info_aat IS TABLE OF info_rt INDEX BY PLS_INTEGER;
    info_aa info_aat;
  BEGIN
    SELECT NAME || '-' || line, text
    BULK COLLECT INTO info_aa FROM user_source
      WHERE UPPER (text) LIKE '%' || UPPER (str) || '%'
      AND NAME != 'VALSTD' AND NAME != 'ERRNUMS';
    DBMS_OUTPUT.PUT_LINE ('Checking for presence of '||
                          str || ':');
    FOR indx IN info_aa.FIRST .. info_aa.LAST LOOP
      DBMS_OUTPUT.PUT_LINE (
          info_aa (indx).NAME|| ',' || info_aa (indx).text);
    END LOOP;
  END find_text_in_code;

  PROCEDURE encap_compliance IS
    SUBTYPE qualified_name_t IS VARCHAR2 (200);
    TYPE refby_rt IS RECORD (NAME qualified_name_t,
        referenced_by qualified_name_t );
    TYPE refby_aat IS TABLE OF refby_rt INDEX BY PLS_INTEGER;
    refby_aa refby_aat;
  BEGIN
    SELECT owner || '.' || NAME refs_table
          , referenced_owner || '.' || referenced_name
          AS table_referenced
    BULK COLLECT INTO refby_aa
      FROM all_dependencies
      WHERE owner = USER
      AND TYPE IN ('PACKAGE', 'PACKAGE BODY',
                   'PROCEDURE', 'FUNCTION')
      AND referenced_type IN ('TABLE', 'VIEW')
      AND referenced_owner NOT IN ('SYS', 'SYSTEM')
     ORDER BY owner, NAME, referenced_owner, referenced_name;
    DBMS_OUTPUT.PUT_LINE ('Programs that reference
                          tables or views');
    FOR indx IN refby_aa.FIRST .. refby_aa.LAST LOOP
      DBMS_OUTPUT.PUT_LINE (refby_aa (indx).NAME || ',' ||
            refby_aa (indx).referenced_by);
    END LOOP;
  END encap_compliance;
END query_code_pkg;
/
```

Finding Coding Information

```
EXECUTE query_code_pkg.encap_compliance
Programs that reference tables or views
OE.PROCESS_CUSTOMERS,OE.CUSTOMERS
OE.PROF_REPORT_UTILITIES,OE.PLSQL_PROFILER_DATA
OE.PROF_REPORT_UTILITIES,OE.PLSQL_PROFILER_LINES_CROSS_RUN
OE.PROF_REPORT_UTILITIES,OE.PLSQL_PROFILER_RUNS
OE.PROF_REPORT_UTILITIES,OE.PLSQL_PROFILER_UNITS
...
PL/SQL procedure successfully completed.
```
1

```
EXECUTE query_code_pkg.find_text_in_code('customers')
Checking for presence of customers:
REPORT_CREDIT-2,  (p_email    customers.cust_last_name%TYPE,
REPORT_CREDIT-3,   p_credit_limit customers.credit_limit%TYPE)
REPORT_CREDIT-5,  TYPE  typ_name IS TABLE OF customers%ROWTYPE
INDEX BY customers.cust_email%TYPE;
REPORT_CREDIT-12,    FOR rec IN  (SELECT * FROM customers WHERE
cust_email IS NOT NULL)
PROCESS_CUSTOMERS-1,PROCEDURE process_customers
...
PL/SQL procedure successfully completed.
```
2

ORACLE

QUERY_CODE_PKG Examples
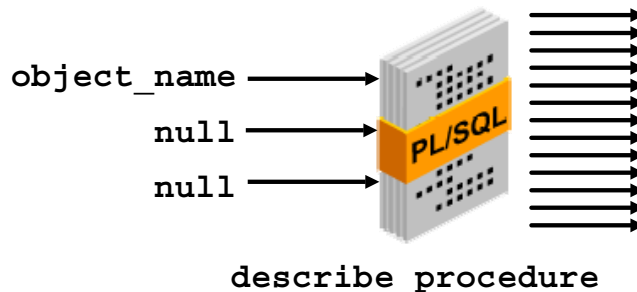
In the first example, the ENCAP_COMPLIANCE procedure displays all PL/SQL code objects that reference a table or view directly. Both the code name and table or view name are listed in the output.

In the second example, the FIND_TEXT_IN_CODE procedure returns all PL/SQL code objects that contain the "customers" text string. The code name, line number, and line are listed in the output.

Using DBMS_DESCRIBE

- **Use it to get information about a PL/SQL object.**
- **It contains one procedure: DESCRIBE_PROCEDURE.**
- **Includes:**
  - **Three scalar IN parameters**
  - **One scalar OUT parameter**
  - **Twelve associative array OUT parameters**

```
object_name ────────▶   ┌──────┐ ──────▶
                        │ PL/SQL│ ──────▶
      null ────────▶    │      │ ──────▶
      null ────────▶    └──────┘ ──────▶

             describe_procedure
```

The DBMS_DESCRIBE Package

You can use the DBMS_DESCRIBE package to find information about your procedures. It contains one procedure, named DESCRIBE_PROCEDURE. This routine accepts the name of the procedure that you are inquiring about. It returns detailed parameter information in a set of associative arrays. The details are numerically coded. You can find the following information from the results returned:

- **Overload:** If overloaded, it holds a value for each version of the procedure.
- **Position:** Position of the argument in the parameter list. 0 is reserved for the RETURN information of a function.
- **Level:** For composite types only; it holds the level of the data type
- **Argument name:** Name of the argument
- **Data type:** A numerically coded value representing a data type
- **Default value:** 0 for no default value, 1 if the argument has a default value
- **Parameter mode:** 0 = IN, 1 = OUT, 2 = IN OUT

Note: This is not the complete list of values returned from the DESCRIBE_PROCEDURE routine. For a complete list, see the *PL/SQL Packages and Types Reference 10g Release 1* reference manual.

Oracle University and En-Sof Informatica E Treinamento Ltda use only

Using `DBMS_DESCRIBE`

Create a package to call the `DBMS_DESCRIBE.DESCRIBE_PROCEDURE` routine:

```
CREATE OR REPLACE PACKAGE use_dbms_describe
IS
  PROCEDURE get_data (p_obj_name VARCHAR2);
END use_dbms_describe;
/
```
1

```
EXEC use_dbms_describe.get_data('ORDERS_APP_PKG.THE_PREDICATE')

Name                                          Mode   Position      Datatype
This is the RETURN data for the function: 1             0             1
P_SCHEMA                                         0             1             1
P_NAME                                           0             2             1

PL/SQL procedure successfully completed.
```
2

The `DESCRIBE_PROCEDURE` Routine

Because the `DESCRIBE_PROCEDURE` returns information about your parameters in a set of associative arrays, it is easiest to define a package to call and handle the information returned from it.

In the first example shown on the slide above, the specification for the `USE_DBMS_DESCRIBE` package is defined. This package holds one procedure, `GET_DATA`. This `GET_DATA` routine calls the `DBMS_DESCRIBE.DESCRIBE_PROCEDURE` routine. The implementation of the `USE_DBMS_DESCRIBE` package is shown on the next page. Note that several associative array variables are defined to hold the values returned via `OUT` parameters from the `DESCRIBE_PROCEDURE` routine. Each of these arrays uses the predefined package types:

```
    TYPE VARCHAR2_TABLE IS TABLE OF VARCHAR2(30)
      INDEX BY BINARY_INTEGER;
    TYPE NUMBER_TABLE IS TABLE OF NUMBER INDEX BY BINARY_INTEGER;
```

In the call to the `DESCRIBE_PROCEDURE` routine, you need to pass three parameters: the name of the procedure that you are inquiring about and two null values. These null values are reserved for future use.

In the second example shown on the slide above, the results are displayed for the parameters of the `ORDERS_APP_PKG.THE_PREDICATE` function. Data type of `1` indicates it is a `VARCHAR2` data type.

The DESCRIBE_PROCEDURE Routine (continued)

Calling DBMS_DESCRIBE.DESCRIBE_PROCEDURE

```
CREATE OR REPLACE PACKAGE use_dbms_describe IS
  PROCEDURE get_data (p_obj_name VARCHAR2);
END use_dbms_describe;
/
CREATE OR REPLACE PACKAGE BODY use_dbms_describe IS
  PROCEDURE get_data (p_obj_name VARCHAR2)
  IS
    v_overload      DBMS_DESCRIBE.NUMBER_TABLE;
    v_position      DBMS_DESCRIBE.NUMBER_TABLE;
    v_level         DBMS_DESCRIBE.NUMBER_TABLE;
    v_arg_name      DBMS_DESCRIBE.VARCHAR2_TABLE;
    v_datatype      DBMS_DESCRIBE.NUMBER_TABLE;
    v_def_value     DBMS_DESCRIBE.NUMBER_TABLE;
    v_in_out        DBMS_DESCRIBE.NUMBER_TABLE;
    v_length        DBMS_DESCRIBE.NUMBER_TABLE;
    v_precision     DBMS_DESCRIBE.NUMBER_TABLE;
    v_scale         DBMS_DESCRIBE.NUMBER_TABLE;
    v_radix         DBMS_DESCRIBE.NUMBER_TABLE;
    v_spare         DBMS_DESCRIBE.NUMBER_TABLE;
  BEGIN
    DBMS_DESCRIBE.DESCRIBE_PROCEDURE
    (p_obj_name, null, null, -- these are the 3 in parameters
     v_overload, v_position, v_level, v_arg_name,
     v_datatype, v_def_value, v_in_out, v_length,
     v_precision, v_scale, v_radix, v_spare, null);
    IF v_in_out.FIRST IS NULL THEN
      DBMS_OUTPUT.PUT_LINE ('No arguments to report.');
    ELSE
      DBMS_OUTPUT.PUT
      ('Name                                     Mode');
      DBMS_OUTPUT.PUT_LINE('  Position    Datatype ');
      FOR i IN v_arg_name.FIRST .. v_arg_name.LAST LOOP
        IF v_position(i) = 0 THEN
          DBMS_OUTPUT.PUT('This is the RETURN data for
          the function: ');
        ELSE
          DBMS_OUTPUT.PUT (
            rpad(v_arg_name(i), LENGTH(v_arg_name(i)) +
                 42-LENGTH(v_arg_name(i)), ' '));
        END IF;
        DBMS_OUTPUT.PUT( '       ' ||
          v_in_out(i) || '           ' || v_position(i) ||
          '              ' || v_datatype(i) );
        DBMS_OUTPUT.NEW_LINE;
      END LOOP;
    END IF;
  END get_data;
END use_dbms_describe;
```

Using `ALL_ARGUMENTS`

Query the `ALL_ARGUMENTS` view to find information about arguments for procedures and functions:

```
SELECT object_name, argument_name, in_out, position, data_type
FROM   all_arguments
WHERE  package_name = 'ORDERS_APP_PKG';


OBJECT_NAME          ARGUMENT_NAME     IN_OUT     POSITION DATA_TYPE
-------------------- ---------------- --------- --------- -----------
THE_PREDICATE        P_NAME           IN               2 VARCHAR2
THE_PREDICATE        P_SCHEMA         IN               1 VARCHAR2
THE_PREDICATE                         OUT              0 VARCHAR2
SET_APP_CONTEXT                       IN               1
SHOW_APP_CONTEXT                      IN               1
```

Using the `ALL_ARGUMENTS` Dictionary View

You can also query the `ALL_ARGUMENTS` dictionary view to find out information about the arguments of procedures and functions to which you have access. Similar to using `DBMS_DESCRIBE`, the `ALL_ARGUMENTS` view returns information in textual rather than numeric form. There is overlap between the two, but there is also unique information to be found both in `DBMS_DESCRIBE` and `ALL_ARGUMENTS`.

In the example shown above, the argument name, mode, position, and data type are returned for the `ORDERS_APP_PKG`. Note the following:
- A position of 1 and a sequence and level of 0 indicates that the procedure has no arguments.
- For a function that has no arguments, it is displayed as a single row for the `RETURN` clause, with a position of 0.
- The argument name for the `RETURN` clause is NULL.
- If programs are overloaded, the `OVERLOAD` column (not shown above) indicates the *N*th overloading; otherwise, it is NULL.
- The `DATA_LEVEL` column (not shown above) value of 0 identifies a parameter as it appears in the program specification.

Using `ALL_ARGUMENTS`

Other column information:

- **Details about the data type are found in the `DATA_TYPE` and `TYPE_` columns.**

- **All arguments in the parameter list are at level 0.**

- **For composite parameters, the individual elements of the composite are assigned levels, starting at 1.**

- **The `POSITION-DATA_LEVEL` column combination is unique only for a level 0 argument (the actual parameter, not its subtypes if it is a composite).**

Using the `ALL_ARGUMENTS` Dictionary View (continued)

The `DATA_TYPE` column holds the generic PL/SQL data type. To find more information about the data type, query the `TYPE_` columns.

- **`TYPE_NAME`:** Holds the name of the type of the argument. If the type is a package local type (that is, it is declared in a package specification), then this column displays the name of the package.
- **`TYPE_SUBNAME`:** Is relevant only for package local types. Displays the name of the type declared in the package identified in the `TYPE_NAME` column. For example, if the data type is a PL/SQL table, you can find out which type of table only by looking at the `TYPE_SUBNAME` column.

Note: The `DEFAULT_VALUE` and `DEFAULT_LENGTH` columns are reserved for future use and do not currently contain information about a parameter's default value. You can use `DBMS_DESCRIBE` to find some default value information. In this package, the parameter `DEFAULT_VALUE` returns 1 if there is a default value; otherwise, it returns 0.

By combining the information from `DBMS_DESCRIBE` and `ALL_ARGUMENTS`, you can find valuable information about parameters, as well as about how your PL/SQL routines are overloaded.

Using
DBMS_UTILITY.FORMAT_CALL_STACK

- **This function returns the formatted text string of the current call stack.**
- **Use it to find the line of code being executed.**

```
EXECUTE third_one
----- PL/SQL Call Stack -----
  object      line  object
  handle    number  name
0x566ce8e0        4  procedure OE.FIRST_ONE
0x5803f7a8        5  procedure OE.SECOND_ONE
0x569c3770        6  procedure OE.THIRD_ONE
0x567ee3d0        1  anonymous block


PL/SQL procedure successfully completed.
```

ORACLE

The DBMS_UTILITY.FORMAT_CALL_STACK Function

Another tool available to you is the FORMAT_CALL_STACK function within the DBMS_UTILITY supplied package. It returns the call stack in a formatted character string. The results shown above were generated based on the following routines:

```
SET SERVEROUT ON
CREATE OR REPLACE PROCEDURE first_one
IS
BEGIN
  dbms_output.put_line(
    substr(dbms_utility.format_call_Stack, 1, 255));
END;
/

CREATE OR REPLACE PROCEDURE second_one
IS
BEGIN
  null;
  first_one;
END;
/
-- continued on next page
```

The `DBMS_UTILITY.FORMAT_CALL_STACK` Function (continued)

```
-- continued from previous page

CREATE OR REPLACE PROCEDURE third_one
IS
BEGIN
  null;
  null;
  second_one;
END;
/
```

The output from the FORMAT_CALL_STACK function shows you the object handle number, line number from where a routine is called, and the routine that is called. Note that the NULL; statements added into the procedures shown are used to emphasize the line number from where the routine is called.

Finding Error Information

DBMS_UTILITY.FORMAT_ERROR_BACKTRACE:

- **Shows you the call stack at the point where an exception is raised.**
- **Returns:**
  - **The backtrace string**
  - **A null string if there are no errors being handled**
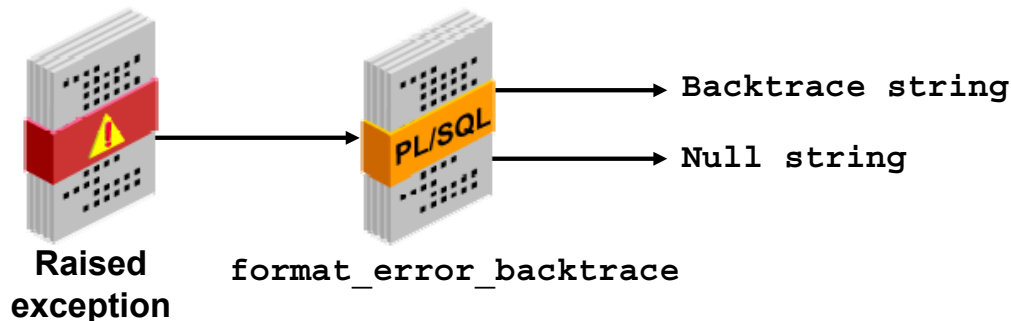
Using DBMS_UTILITY.FORMAT_ERROR_BACKTRACE

You can use this function to display the call stack at the point where an exception was raised, even if the procedure is called from an exception handler in an outer scope. The output returned is similar to the output of the SQLERRM function, but not subject to the same size limitation.

Using DBMS_UTILITY.FORMAT_ERROR_STACK

You can use this function to format the current error stack. It can be used in exception handlers to view the full error stack. The function returns the error stack, up to 2,000 bytes.

Finding Error Information

```
CREATE OR REPLACE PROCEDURE top_with_logging IS
  -- NOTE: SQLERRM in principle gives the same info
  -- as format_error_stack.
  -- But SQLERRM is subject to some length limits,
  -- while format_error_stack is not.
BEGIN
  P5(); -- this procedure, in turn, calls others,
        -- building a stack. P0 contains the exception
EXCEPTION
  WHEN OTHERS THEN
    log_errors ( 'Error_Stack...' || CHR(10) ||
      DBMS_UTILITY.FORMAT_ERROR_STACK() );
    log_errors ( 'Error Backtrace...' || CHR(10) ||
      DBMS_UTILITY.FORMAT_ERROR_BACKTRACE() );
    DBMS_OUTPUT.PUT_LINE ( '----------' );
END top_with_logging;
/
```

ORACLE

Using FORMAT_ERROR_STACK and FORMAT_ERROR_BACKTRACE

To show you the functionality of the FORMAT_ERROR_STACK and
FORMAT_ERROR_BACKTRACE functions, a TOP_WITH_LOGGING procedure is created. This
procedure calls the LOG_ERRORS procedure and passes to it the results of the
FORMAT_ERROR_STACK and FORMAT_ERROR_BACKTRACE functions.

The LOG_ERRORS procedure is shown on the next page.

Finding Error Information

```
CREATE OR REPLACE PROCEDURE log_errors ( i_buff IN VARCHAR2 ) IS
  g_start_pos PLS_INTEGER := 1;
  g_end_pos   PLS_INTEGER;
  FUNCTION output_one_line RETURN BOOLEAN IS
  BEGIN
    g_end_pos := INSTR ( i_buff, CHR(10), g_start_pos );
    CASE g_end_pos > 0
      WHEN TRUE THEN
        DBMS_OUTPUT.PUT_LINE ( SUBSTR ( i_buff,
                               g_start_pos, g_end_pos-g_start_pos ));
        g_start_pos := g_end_pos+1;
        RETURN TRUE;
      WHEN FALSE THEN
        DBMS_OUTPUT.PUT_LINE ( SUBSTR ( i_buff, g_start_pos,
                               (LENGTH(i_buff)-g_start_pos)+1 ));
        RETURN FALSE;
    END CASE;
  END output_one_line;
BEGIN
  WHILE output_one_line() LOOP NULL;
  END LOOP;
END log_errors;
```

The `LOG_ERRORS` Example

This procedure takes the return results of the FORMAT_ERROR_STACK and
FORMAT_ERROR_BACKTRACE functions as an IN string parameter, and reports it back to you
using DBMS_OUTPUT.PUT_LINE. The LOG_ERRORS procedure is called twice from the
TOP_WITH_LOGGING procedure. The first call passes the results of FORMAT_ERROR_STACK
and the second procedure passes the results of FORMAT_ERROR_BACKTRACE .

Note: You could use UTL_FILE instead of DBMS_OUTPUT to write and format the results to a
file.

The `LOG_ERRORS` Example (continued)

Next, several procedures are created and one procedure calls another, so that a stack of procedures is built. The `P0` procedure raises a zero divide exception when it is invoked. The call stack is:

```
TOP_WITH_LOGGING > P5 > P4 > P3 > P2 > P1 > P0
```

```
SET DOC OFF
SET FEEDBACK OFF
SET ECHO OFF

CREATE OR REPLACE PROCEDURE P0 IS
  e_01476 EXCEPTION;
  pragma exception_init ( e_01476, -1476 );
BEGIN
  RAISE e_01476;  -- this is a zero divide error
END P0;
/
CREATE OR REPLACE PROCEDURE P1 IS
BEGIN
  P0();
END P1;
/
CREATE OR REPLACE PROCEDURE P2 IS
BEGIN
  P1();
END P2;
/
CREATE OR REPLACE PROCEDURE P3 IS
BEGIN
  P2();
END P3;
/
CREATE OR REPLACE PROCEDURE P4 IS
  BEGIN P3();
END P4;
/
CREATE OR REPLACE PROCEDURE P5 IS
  BEGIN P4();
END P5;
/
CREATE OR REPLACE PROCEDURE top IS
BEGIN
  P5(); -- this procedure is used to show the results
        -- without using the TOP_WITH_LOGGING routine.
END top;
/
SET FEEDBACK ON
```

Finding Error Information

Results:

```
EXECUTE top_with_logging
Error_Stack...
ORA-01476: divisor is equal to zero
Error_Backtrace...
ORA-06512: at "OE.P0", line 5
ORA-06512: at "OE.P1", line 3
ORA-06512: at "OE.P2", line 3
ORA-06512: at "OE.P3", line 3
ORA-06512: at "OE.P4", line 2
ORA-06512: at "OE.P5", line 2
ORA-06512: at "OE.TOP_WITH_LOGGING", line 7
----------
```

ORACLE

Finding Error Information Results

The results from executing the TOP_WITH_LOGGING procedure is shown. Note that the error stack displays the exception encountered. The backtrace information traces the flow of the exception to its origin.

If you execute the TOP procedure without using the TOP_WITH_LOGGING procedure, these are the results:

```
EXECUTE top
BEGIN top; END;
*
ERROR at line 1:
ORA-01476: divisor is equal to zero
ORA-06512: at "OE.P0", line 5
ORA-06512: at "OE.P1", line 3
ORA-06512: at "OE.P2", line 3
ORA-06512: at "OE.P3", line 3
ORA-06512: at "OE.P4", line 2
ORA-06512: at "OE.P5", line 2
ORA-06512: at "OE.TOP", line 3
ORA-06512: at line 1
```

Note that the line number reported is misleading.

Tracing PL/SQL Execution

Tracing PL/SQL execution provides you with a better understanding of the program execution path, and is possible by using the `dbms_trace` package.

Enable specific subprograms for tracing (optional)

↓

Start tracing session

Trace data

Trace data

↓

Run application to be traced

↓

Stop tracing session
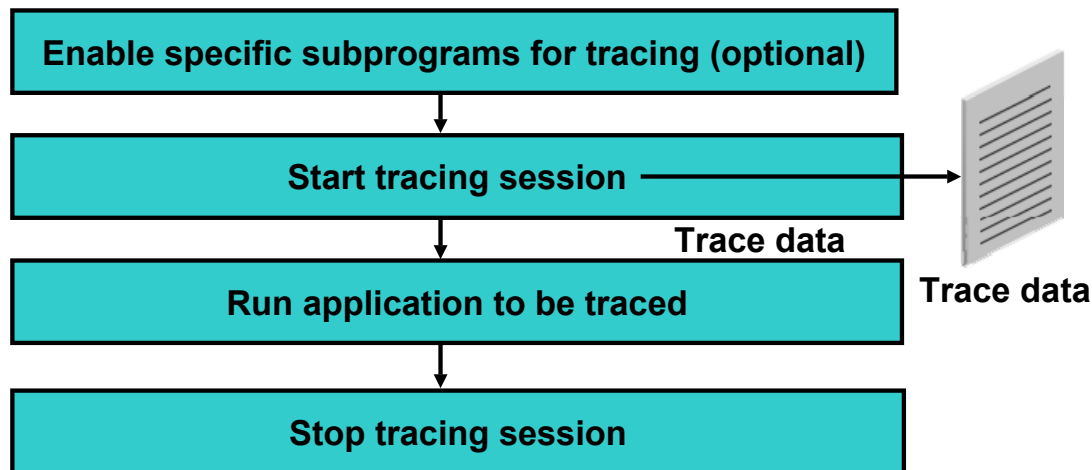
Tracing PL/SQL Execution

In large and complex PL/SQL applications, it can sometimes become difficult to keep track of subprogram calls when a number of them call each other. By tracing your PL/SQL code, you can get a clearer idea of the paths and order in which your programs execute.

While a facility to trace your SQL code has been around for a while, Oracle now provides an API for tracing the execution of PL/SQL programs on the server. You can use the Trace API, implemented on the server as the `dbms_trace` package, to trace PL/SQL subprogram code.

Note: You cannot use PL/SQL tracing with the multithreaded server (MTS).

Tracing PL/SQL Execution

The `dbms_trace` package contains:

- `set_plsql_trace` (*trace_level* `INTEGER`)
- `clear_plsql_trace`
- `plsql_trace_version`

The `dbms_trace` Programs

`dbms_trace` provides subprograms to start and stop PL/SQL tracing in a session. The trace data is collected as the program executes, and it is written out to data dictionary tables.

Procedure	Description
`set_plsql_trace`	Start tracing data dumping in a session (You provide the trace level at which you want your PL/SQL code traced as an `IN` parameter.)
`clear_plsql_trace`	Stops trace data dumping in a session
`plsql_trace_version`	Returns the version number of the trace package as an out parameter

A typical trace session involves:

- Enabling specific subprograms for trace data collection (optional)
- Starting the PL/SQL tracing session (`dbms_trace.set_plsql_trace`)
- Running the application that is to be traced
- Stopping the PL/SQL tracing session (`dbms_trace.clear_plsql_trace`)

Tracing PL/SQL Execution

- **Using `set_plsql_trace`, select a trace level to identify how to trace calls, exceptions, SQL, and lines of code.**
- **Trace-level constants:**
  - `trace_all_calls`
  - `trace_enabled_calls`
  - `trace_all_sql`
  - `trace_enabled_sql`
  - `trace_all_exceptions`
  - `trace_enabled_exceptions`
  - `trace_enabled_lines`
  - `trace_all_lines`
  - `trace_stop`
  - `trace_pause`
  - `trace_resume`

ORACLE

Specifying a Trace Level

During the trace session, there are two levels that you can specify to trace calls, exceptions, SQL, and lines of code.

Trace Calls

- **Level 1:** Trace all calls. This corresponds to the constant `trace_all_calls`.
- **Level 2:** Trace calls to enabled program units only. This corresponds to the constant `trace_enabled_calls`.

Trace Exceptions

- **Level 1:** Trace all exceptions. This corresponds to `trace_all_exceptions`.
- **Level 2:** Trace exceptions raised in enabled program units only. This corresponds to `trace_enabled_exceptions`.

Trace SQL

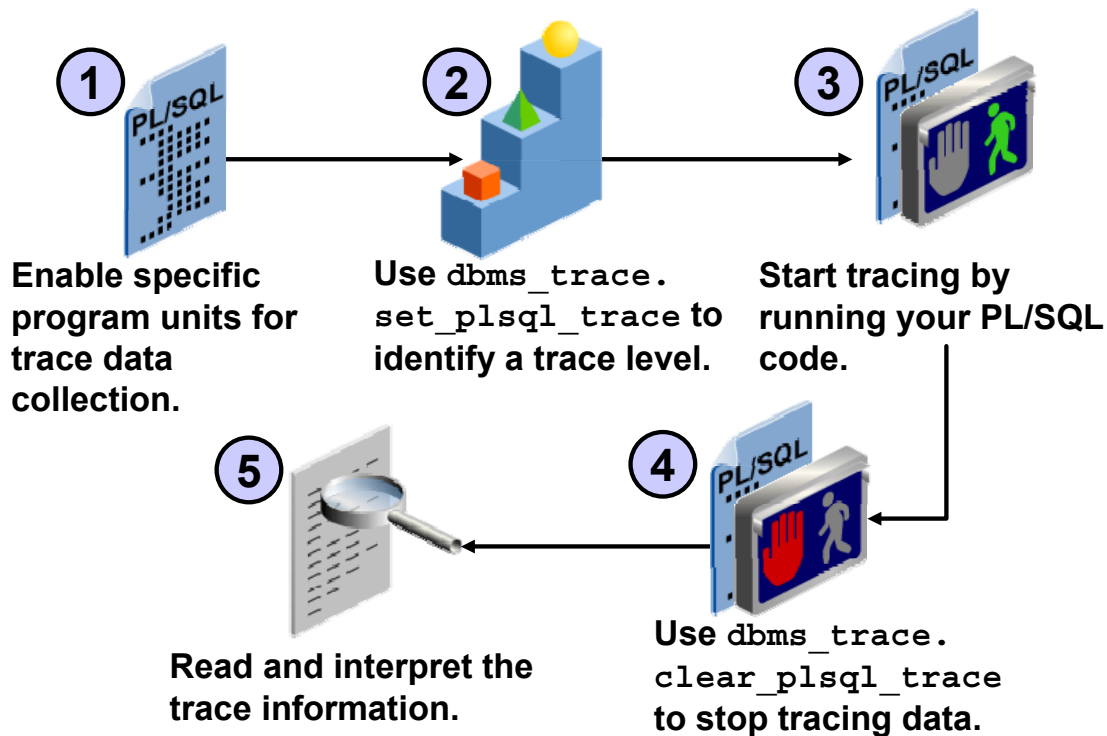- **Level 1:** Trace all SQL. This corresponds to the constant `trace_all_sql`.
- **Level 2:** Trace SQL in enabled program units only. This corresponds to the constant `trace_enabled_sql`.

Trace Lines

- **Level 1:** Trace all lines. This corresponds to the constant `trace_all_lines`.
- **Level 2:** Trace lines in enabled program units only. This corresponds to the constant `trace_enabled_lines`.

Tracing PL/SQL: Steps

1. Enable specific program units for trace data collection.
2. Use `dbms_trace.set_plsql_trace` to identify a trace level.
3. Start tracing by running your PL/SQL code.
4. Use `dbms_trace.clear_plsql_trace` to stop tracing data.
5. Read and interpret the trace information.

ORACLE

Steps to Trace PL/SQL Code

There are five steps to trace PL/SQL code using the `dbms_trace` package:

1. Enable specific program units for trace data collection.
2. Use `dbms_trace.set_plsql_trace` to identify a trace level.
3. Run your PL/SQL code.
4. Use `dbms_trace.clear_plsql_trace` to stop tracing data.
5. Read and interpret the trace information.

The next few pages demonstrate the steps to accomplish PL/SQL tracing.

Step 1: Enable Specific Subprograms

Enable specific subprograms with one of the two methods:

- **Enable a subprogram by compiling it with the debug option:**

```
ALTER SESSION SET PLSQL_DEBUG=true;
```

```
CREATE OR REPLACE ....
```

- **Recompile a specific subprogram with the debug option:**

```
ALTER [PROCEDURE | FUNCTION | PACKAGE]
<subprogram-name> COMPILE DEBUG [BODY];
```

Step 1: Enable Specific Subprograms

Profiling large applications may produce a huge volume of data that can be difficult to manage. Before turning on the trace facility, you have the option to control the volume of data collected by enabling a specific subprogram for trace data collection. You can enable a subprogram by compiling it with the debug option. You can do this in one of two ways:

- Enable a subprogram by compiling it with the ALTER SESSION debug option, then compile the program unit by using CREATE OR REPLACE syntax:

```
ALTER SESSION SET PLSQL_DEBUG = true;
CREATE OR REPLACE ...
```

- Alternatively, recompile a specific subprogram with the debug option:

```
ALTER [PROCEDURE | FUNCTION | PACKAGE]
            <subprogram-name> COMPILE DEBUG [BODY];
```

Note: The second method cannot be used for anonymous blocks.

Enabling specific subprograms allows you to:

- Limit and control the amount of trace data, especially in large applications.
- Obtain additional trace information that is otherwise not available. For example, during the tracing session, if a subprogram calls another subprogram, the name of the called subprogram gets included in the trace data if the calling subprogram was enabled by compiling it in debug mode.

Steps 2 and 3: Identify a Trace Level and Start Tracing

- **Specify the trace level by using dbms_trace.set_plsql_trace:**

```
EXECUTE DBMS_TRACE.SET_PLSQL_TRACE -
   (tracelevel1 + tracelevel2 ...)
```

- **Execute the code to be traced:**

```
EXECUTE my_program
```

Steps 2 and 3: Specify a Trace Level and Start Tracing

To trace PL/SQL code execution by using dbms_trace, follow these steps:
- Start the trace session using the syntax in the slide. For example:
```
EXECUTE –
DBMS_TRACE.SET_PLSQL_TRACE(DBMS_TRACE.trace_all_calls)
```

Note:
- To specify additional trace levels in the argument, use the "+" sign between each trace level value.
- Execute the PL/SQL code. The trace data gets written to either the Oracle server trace file or to the data dictionary views.

Step 4: Turn Off Tracing

Remember to turn tracing off by using the
`dbms_trace.clear_plsql_trace` **procedure.**

```
EXECUTE DBMS_TRACE.CLEAR_PLSQL_TRACE
```

Step 4: Turn Off Tracing

When you have completed tracing the PL/SQL program unit, turn tracing off by executing
`dbms_trace.clear_plsql_trace`. This stops any further writing to the trace file.

To avoid the overhead of writing the trace information, it is recommended that you turn off the
tracing when you are not using it.

Step 5: Examine the Trace Information

Examine the trace information:

- **Call tracing writes out the program unit type, name, and stack depth.**
- **Exception tracing writes out the line number.**

Step 5: Examine the Trace Information

- Lower trace levels supersede higher levels when tracing is activated for multiple tracing levels.
- If tracing is requested only for enabled subprograms, and if the current subprogram is not enabled, then no trace data is written.
- If the current subprogram is enabled, then call tracing writes out the subprogram type, name, and stack depth.
- If the current subprogram is not enabled, then call tracing writes out the subprogram type, line number, and stack depth.
- Exception tracing writes out the line number. Raising the exception shows information about whether the exception is user-defined or predefined and, in the case of predefined exceptions, the exception number.

Note: An enabled subprogram is compiled with the debug option.

`plsql_trace_runs` and `plsql_trace_events`

- **Trace information is written to the following dictionary views:**
  - `plsql_trace_runs` **dictionary view**
  - `plsql_trace_events` **dictionary view**
- **Run the `tracetab.sql` script to create the dictionary views.**
- **You need privileges to view the trace information in the dictionary views.**

ORACLE

The `plsql_trace_runs` and `plsql_trace_events` Dictionary Views

All trace information is written to the dictionary views `plsql_trace_runs` and `plsql_trace_events`. These views are created (typically by a DBA) by running the `tracetab.sql` script. After the script is run, you need the `SELECT` privilege to view information from these dictionary views.

Note: With the Oracle release 8.1.6 and later, the trace information is written to the dictionary views. Prior to release 8.1.6, trace files were generated and trace information was written to the file. The location of this file is determined by the `USER_DUMP_DEST` initialization parameter. A file with a `.trc` extension is generated during the tracing.

`plsql_trace_runs` and `plsql_trace_events`

```
SELECT proc_name, proc_line,
       event_proc_name, event_comment
FROM sys.plsql_trace_events
WHERE event_proc_name = 'P5'
OR PROC_NAME = 'P5';
```

```
PROC_NAME   PROC_LINE EVENT_PROC_NAME  EVENT_COMMENT
---------- ---------- ---------------- ---------------
P5                  1                  Procedure Call
P4                  1 P5               Procedure Call

2 rows selected.
```

Query the `plsql_trace_runs` and `plsql_trace_events` Views

Use the dictionary views `plsql_trace_runs` and `plsql_trace_events` to view the trace information generated by using the `dbms_trace` facility. `plsql_trace_runs` holds generic information about traced programs such as the date, time, owner, and name of the traced stored program. `dbms_trace_events` holds more specific information about the traced subprograms.

Oracle University and En-Sof Informatica E Treinamento Ltda use only

Profiling PL/SQL Applications

You can use profiling to evaluate performance and identify areas that need improvement.

- **Count the number of times each line was executed.**
- **Determine how much time was spent on each line.**
- **Access the gathered information stored in database tables, and can be viewed at any desired level of granularity.**
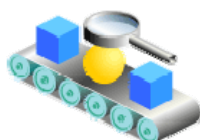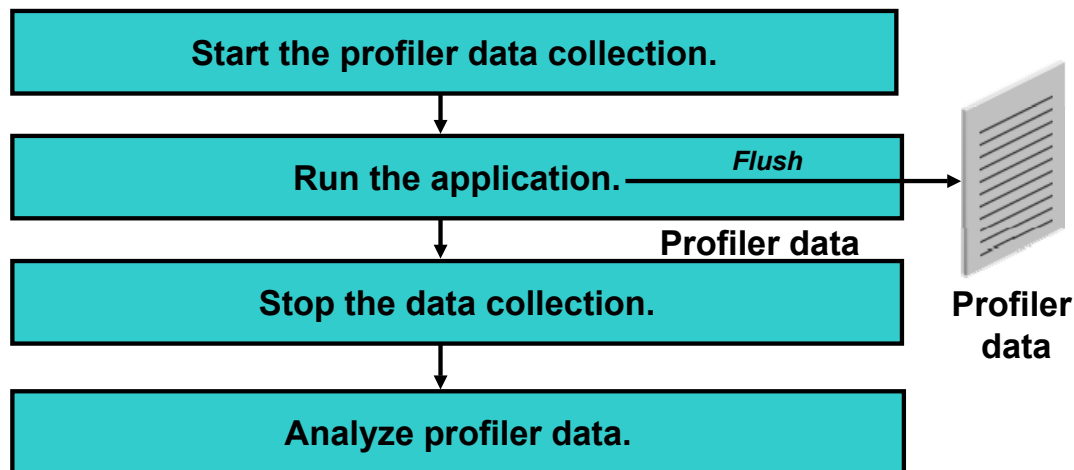
ORACLE

Profiling PL/SQL Applications

PL/SQL provides a tool called the Profiler that can be used to determine the execution time profile (or run-time behavior) of applications. The Profiler can be used to figure out which part of a particular application is running slowly. Such a tool is crucial in identifying performance bottlenecks. It can help you focus your efforts on improving the performance of only the relevant PL/SQL components, or, even better, the particular program segments where a lot of execution time is being spent.

The Profiler provides functions for gathering "profile" statistics, such as the total number of times each line was executed; time spent executing each line; and minimum and maximum duration spent on execution of a given line of code. For example, you can generate profiling information for all named library units used in a single session. This information is stored in database tables that can be queried later.

Third-party vendors can use the profiling API to build graphical, customizable tools. You can use Oracle 10*g*'s sample (demo) text-based report writer to gather meaningful data about their applications. The script is called `profrep.sql` and you can find it in your `Oracle_home/PLSQL/demo` directory. You can use the profiling API to analyze the performance of your PL/SQL applications and to locate bottlenecks. You can then use the profile information to appropriately tune your application.

Profiling PL/SQL Applications

Use `DBMS_PROFILER` to profile existing PL/SQL applications and to identify performance bottlenecks.

Start the profiler data collection.

Run the application. *Flush*

Profiler data

Stop the data collection.

Profiler data

Analyze profiler data.

Profiling PL/SQL Applications (continued)

The profiler API is implemented as a PL/SQL package, `DBMS_PROFILER`, which provides services for collecting and persistently storing PL/SQL profiler data.

Note: To set up profiling, two scripts need to be run. The `profload.sql` script is run under `SYS`. The `proftab.sql` script creates the profile dictionary tables. Run this script in the schema under which you want to collect profiling statistics.

Profiling PL/SQL Applications

The `dbms_profiler` package contains:

- **START_PROFILER**
- **STOP_PROFILER**
- **FLUSH_DATA**
- **PAUSE_PROFILER**
- **RESUME_PROFILER**
- **GET_VERSION**
- **INTERNAL_VERSION_CHECK**

ORACLE

Profiling PL/SQL Applications (continued)

Routine	Description
START_PROFILER function	Starts profiler data collection in the user's session
STOP_PROFILER function	Stops profiler data collection in the user's session
FLUSH_DATA function	Flushes profiler data collected in the user's session
PAUSE_PROFILER function	Pauses profiler data collection
RESUME_PROFILER function	Resumes profiler data collection
GET_VERSION procedure	Gets the version of this API
INTERNAL_VERSION_ CHECK function	Verifies that this version of the DBMS_PROFILER package can work with the implementation in the database

Profiling PL/SQL: Steps

1 Start profiler data collection in the run.

2 Execute PL/SQL code for which profiler and code coverage is required.

3 Flush data to the profiler tables.

5 Analyze the data collected.
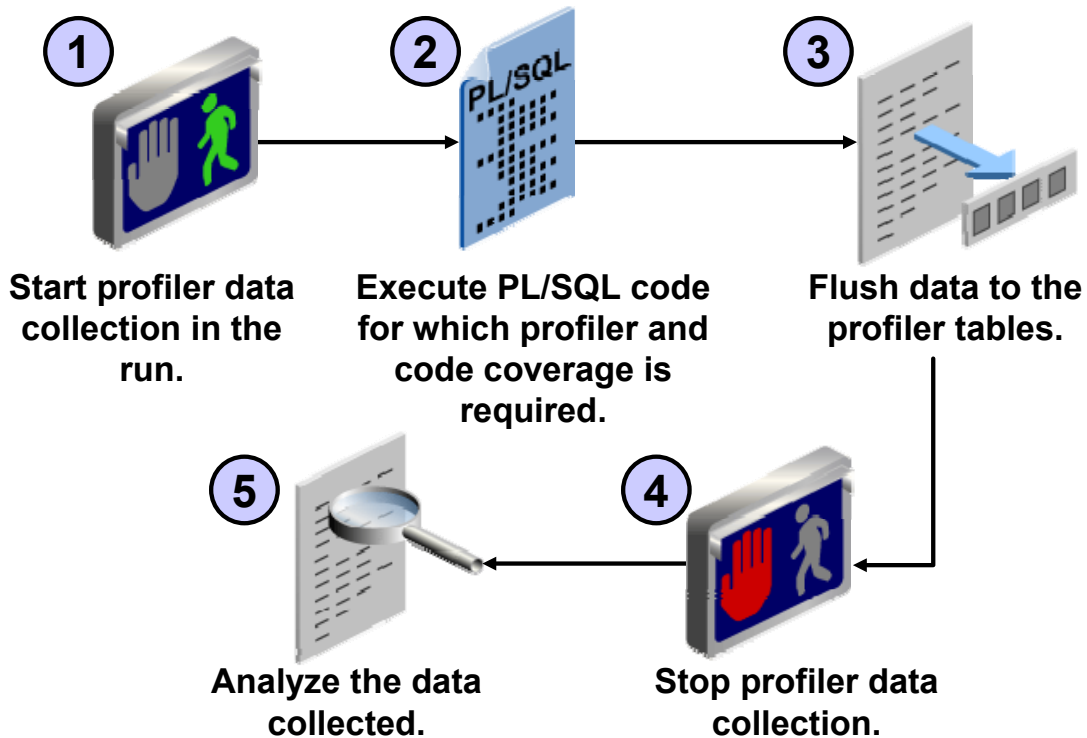
4 Stop profiler data collection.

Steps to Profile PL/SQL Code

To profile PL/SQL code by using the `dbms_profiler` package, perform the following steps:
1. Start the profiler data collection by using `dbms_profiler.start_run`.
2. Execute the application that you are benchmarking.
3. Flush the data collected to the profiler tables by using `dbms_profiler.flush_data`.
4. Stop the profiler data collection by using `dbms_profiler.stop_run`.

Read and interpret the profiler information in the profiler tables:
- `PLSQL_PROFILER_RUNS`
- `PLSQL_PROFILER_UNITS`
- `PLSQL_PROFILER_DATA`

Profiling Example

```
CREATE OR REPLACE PROCEDURE my_profiler
(p_comment1 IN VARCHAR2, p_comment2 IN VARCHAR2)
IS
  v_return_code      NUMBER;
BEGIN
--start the profiler
  v_return_code:=DBMS_PROFILER.START_PROFILER(p_comment1, p_comment2);
  dbms_output.put_line ('Result from START: '||v_return_code);

-- now run a program...
  query_code_pkg.find_text_in_code('customers');

--flush the collected data to the dictionary tables
  v_return_code := DBMS_PROFILER.FLUSH_DATA;
  dbms_output.put_line ('Result from FLUSH: '||v_return_code);

--stop profiling
  v_return_code := DBMS_PROFILER.STOP_PROFILER;
  dbms_output.put_line ('Result from STOP: '||v_return_code);
END;
/
```

Running the Profiler

The my_profiler sample procedure shown starts the profiler, runs an application, flushes the data collected from the profiler to the dictionary tables, and stops the profiler. The functions start_profiler, flush_data, and stop_profiler return a numeric value indicating whether the function ran successfully. A return value of 0 indicates success.

Return Code	Meaning
0	Function ran successfully.
1	A subprogram was called with an incorrect parameter.
2	Data flush operation failed. Check whether the profiler tables have been created, are accessible, and that there is adequate space.
-1	There is a mismatch between package and database implementation.

start_profiler accepts two run comments as parameters. These two run comments default to the sysdate and null if they are not specified.

Profiling Example

```
EXECUTE my_profiler('Benchmark: 1', 'This is the first run!')
Result from START: 0
...
Result from FLUSH: 0
Result from STOP: 0

PL/SQL procedure successfully completed.
```

```
SELECT runid, run_owner, run_date, run_comment,
       run_comment1, run_total_time
FROM   plsql_profiler_runs;

  RUNID    RUN_OWNER  RUN_DATE  RUN_COMMENT  RUN_COMMEN RUN_TOTAL_TIME
---------- ---------- --------- ------------ ---------- --------------
         1 OE         23-MAY-04 Benchmark: 1 This is th     7.2632E+10
                                             e first ru
                                             n!
```

Examining the Results

The code shown in the slide shows some basic statistics. The query retrieves the RUNID, which
can be used to find more information.

Profiling Example

- **Find the `runid` and `unit_number`:**

```
SELECT runid, unit_number, unit_type, unit_owner, unit_name
FROM   plsql_profiler_units inner JOIN plsql_profiler_runs
USING ( runid );
```

```
RUNID UNIT_NUMBER UNIT_TYPE       UNIT_OWNER   UNIT_NAME
----- ----------- -------------- ------------ ------------
    1           1 PROCEDURE       OE           MY_PROFILER
    1           2 PACKAGE BODY    OE           QUERY_CODE_PKG
```

- **Use the `runid` and `unit_number` to view the timings per line of code:**

```
SELECT line#, total_occur, total_time, min_time, max_time
FROM   plsql_profiler_data
WHERE  runid = 1 AND unit_number = 2;
```

Profiling Example

Query from the `PLSQL_PROFILER_DATA` table to view the timings per line of code executed.

| LINE#	TOTAL_OCCUR	TOTAL_TIME	MIN_TIME	MAX_TIME
8	1	225494518	225494518	225494518
12	1	2948418	2948418	2948418
13	0	0	0	0
14	1	553980	553980	553980
15	0	0	0	0
16	1	703999	703999	703999
17	0	0	0	0
19	1	1036723	1036723	1036723
21	1	844140290	844140290	844140290
24	1	2911542	2911542	2911542
25	1	317638	317638	317638
LINE#	TOTAL_OCCUR	TOTAL_TIME	MIN_TIME	MAX_TIME
30	1	3710247	3710247	3710247

12 rows selected.

Summary

In this lesson, you should have learned how to:

- **Use the dictionary views and supplied packages to get information about your PL/SQL application code**
- **Trace a PL/SQL application by using `DBMS_TRACE`**
- **Profile a PL/SQL application by using `DBMS_PROFILE`**

| Supplied packages | Dictionary views | Code analysis | Interpret information |
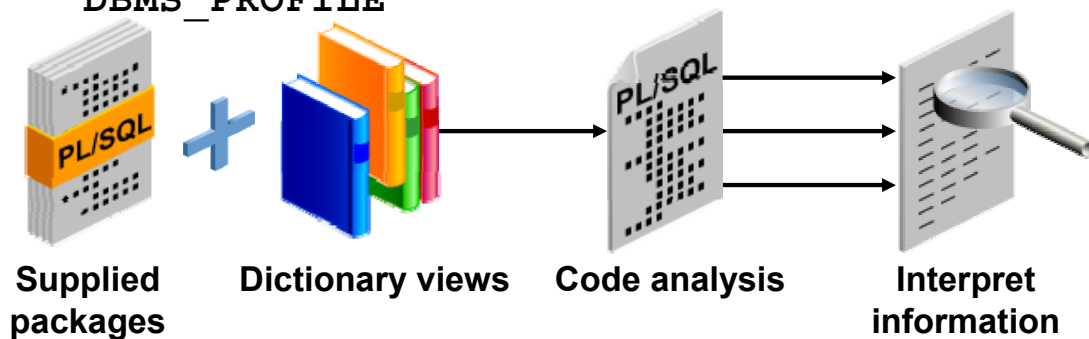
Summary

In this lesson, you learned how to use the dictionary views and supplied PL/SQL packages to analyze your PL/SQL applications.

Practice Overview

This practice covers the following topics:
- **Tracing components in your `OE` application.**
- **Profiling components in your `OE` application.**

Practice Overview

Using the `OE` application that you have created, write code to analyze your application.
- Trace components in your `OE` application
- Profile components in your `OE` application

For detailed instructions on performing this practice, see Appendix A, "Practice Solutions."

Practice 8

In this exercise, you will profile the CREDIT_CARD_PKG package created in an earlier lesson.

1. Run the `lab_08_01.sql` script to create the `CREDIT_CARD_PKG` package.

2. Run the `proftab.sql` script to create the profile tables under your schema.

3. Create a `MY_PROFILER` procedure to:
    -   Start the profiler
    -   Run the application
        ```
        EXECUTE credit_card_pkg.update_card_info –
                (130, 'AM EX', 121212121212)
        ```
    -   Flush the profiler data
    -   Stop the profiler

4. Execute the `MY_PROFILER` procedure.

5. Analyze the results of profiling in the `PLSQL_PROFILER` tables.

    In this exercise, you will trace the `CREDIT_CARD_PKG` package.

6. Enable the `CREDIT_CARD_PKG` for tracing by using the `ALTER` statement with the `COMPILE DEBUG` option.

7. Start the trace session and trace all calls.

8. Run the `credit_card_pkg.update_card_info` procedure with the following data:
    ```
    EXECUTE credit_card_pkg.update_card_info –
                (135, 'DC', 987654321)
    ```

9. Disable tracing.

10. Examine the trace information by querying the trace tables.

```
PROC_NAME            PROC_LINE EVENT_PROC_NAME      EVENT_COMMENT
----------------- ---------- ------------------ ----------------------
CUST_CARD_INFO            4 UPDATE_CARD_INFO     Procedure Call
                            CUST_CARD_INFO       PL/SQL Internal Call
UPDATE_CARD_INFO         31 CUST_CARD_INFO       Return from procedure
                                                 call
                          1 UPDATE_CARD_INFO     Return from procedure
                                                 call
```

A
Practice Solutions

Practice 1: Solutions

PL/SQL Basics

1. What are the four key areas of the basic PL/SQL block? What happens in each area?
   Header section: Names the program unit and identifies it as a procedure, function, or package; also identifies any parameters that the code may use
   Declarative section: Area used to define variables, constants, cursors, and exceptions; starts with the keyword IS or AS Executable section: Main processing area of the PL/SQL program; starts with the keyword BEGIN Exception handler section: Optional error handling section; starts with the keyword EXCEPTION

2. What is a variable and where is it declared?
   Variables are used to store data during PL/SQL block execution.
   You can declare variables in the declarative part of any PL/SQL block, subprogram, or package. Declarations allocate storage space for a value, specify its data type, and name the storage location so that you can reference it. Declarations can also assign an initial value and impose the NOT NULL constraint on the variable.
   Syntax: `variable_name datatype[(size)][:= initial_value];`

3. What is a constant and where is it declared?
   Constants are variables that never change. Constants are declared and assigned a value in the declarative section, before the executable section.
   Syntax: `constant_name CONSTANT datatype[(size)] := initial_value;`

4. What are the different modes for parameters and what does each mode do?
   There are three parameter modes: IN, OUT, and IN OUT. IN is the default and it means a value is passed into the program. The OUT mode indicates that the subprogram is passing a value generated in the subprogram out to the calling environment. The IN OUT mode means that a value is passed into the subprogram. The subprogram may change the value and pass the value out to the calling environment.

5. How does a function differ from a procedure?
   A function must execute a RETURN statement that returns a value. Functions are called differently than procedures. They are called as an expression embedded within another command. Procedures are called as statements.

6. What are the two main components of a PL/SQL package?
   The package body and package specification

   a. In what order are they defined?
      First the package specification and then the package body

   b. Are both required?
      No, only a package specification is required. A specification can exist without a body, but a body cannot exist as valid without the specification.

Practice 1: Solutions (continued)

7. How does the syntax of a `SELECT` statement used within a PL/SQL block differ from a `SELECT` statement issued in SQL*Plus?
   The `INTO` clause is required with a `SELECT` statement that is in a PL/SQL subprogram.

8. What is a record?
   A record is a composite type that has internal components, which can be manipulated individually. Use the `RECORD` data type to treat related but dissimilar data as a logical unit.

9. What is an index-by table?
   Index-by tables are a data structure declared in a PL/SQL block. It is similar to an array and made of two components, the index and the data field. The data field is a column of a scalar or record data type, which stores the `INDEX BY` table elements.

10. How are loops implemented in PL/SQL?
    Looping constructs are used to repeat a statement or sequence of statements multiple times. PL/SQL has three looping constructs:

    - **Basic loops that perform repetitive actions without overall conditions**

    - **`FOR` loops that perform iterative control of actions based on a count**

    - **`WHILE` loops that perform iterative control of actions based on a condition**

11. How is branching logic implemented in PL/SQL?
    You can change the logical flow of statements within the PL/SQL block with a number of control structures. Branching logic is implemented within PL/SQL by using the conditional `IF` statement or `CASE` expressions.

Cursor Basics

12. What is an explicit cursor?
    The Oracle server uses work areas, called private SQL areas, to execute SQL statements and to store processing information. You can use PL/SQL cursors to name a private SQL area and access its stored information. Use explicit cursors to individually process each row returned by a multiple-row `SELECT` statement.

13. Where do you define an explicit cursor?
    A cursor is defined in the declarative section.

14. Name the five steps for using an explicit cursor.
    `Declare`, `Open`, `Fetch`, `Test for existing rows`, and `Close`

15. What is the syntax used to declare a cursor?
    `CURSOR cursor_name IS SELECT_statement`

Oracle Database 10g: Advanced PL/SQL   A-3

Practice 1: Solutions (continued)

16. What does the `FOR UPDATE` clause do within a cursor definition?
    The `FOR UPDATE` clause locks the rows selected in the `SELECT` statement definition of the cursor.

17. What command opens an explicit cursor?
    `OPEN cursor_name;`

18. What command closes an explicit cursor?
    `CLOSE cursor_name;`

19. Name five implicit actions that a cursor `FOR` loop provides.
    Declares a record structure to match the select list of the cursor; opens the cursor, fetches from the cursor, exits the loop when the fetch returns no row, and closes the cursor

20. Describe what the following cursor attributes do:
    `%ISOPEN`: Returns a Boolean value indicating whether the cursor is open
    `%FOUND`: Returns a Boolean value indicating whether the last fetch returned a value
    `%NOTFOUND`: Returns a Boolean value indicating whether the last fetch did not return a value
    `%ROWCOUNT`: Returns an integer indicating the number of rows fetched so far

Exceptions

21. An exception occurs in your PL/SQL block which is enclosed in another PL/SQL block. What happens to this exception?
    Control is passed to the exception handler. If the exception is handled in the inner block, processing continues to the outer block. If the exception is not handled in the inner block, an exception is raised in the outer block and control is passed to the exception handler of the outer block. If neither the inner nor the outer block traps the exception, the program ends unsuccessfully.

22. An exception handler is mandatory within a PL/SQL subprogram. (True/False)
    False

23. What syntax do you use in the exception handler area of a subprogram?
    ```
    EXCEPTION
            WHEN named_exception THEN
                statement[s];

            WHEN others THEN
                statement[s];
    END;
    ```

Practice 1: Solutions (continued)

24. How do you code for a `NO_DATA_FOUND` error?
    EXCEPTION
    **   WHEN no_data_found THEN**
    **   statement[s];**
    END;

25. Name three types of exceptions.
    User-defined, Oracle server predefined, and Oracle server non-predefined

26. To associate an exception identifier with an Oracle error code, what pragma do you use and where?
    Use the PRAGMA EXCEPTION_INIT and place the PRAGMA EXCEPTION_INIT in the declarative section.

27. How do you explicitly raise an exception?
    Use the RAISE statement or the raise_application_error procedure.

28. What types of exceptions are implicitly raised?
    All Oracle server exceptions (predefined and non-predefined) are automatically raised.

29. What does the `RAISE_APPLICATION_ERROR` procedure do?
    Enables you to issue user-defined error messages from subprograms.

Dependencies

30. Which objects can a procedure or function directly reference?
    Table, view, sequence, procedure, function, package specification, object specification, and collection type

31. What are the two statuses that a schema object can have and where are they recorded?
    The user_objects dictionary view contains a column called status. Its values are VALID and INVALID.

32. The Oracle server automatically recompiles invalid procedures when they are called from the same _____. To avoid compile problems with remote database calls, we can use the _____ model instead of the timestamp model.
    database
    signature

33. What data dictionary contains information on direct dependencies?
    user_dependencies

34. What script do you run to create the views deptree and ideptree?
    You use the utldtree.sql script.

35. What does the `deptree_fill` procedure do and what are the arguments that you need to provide?

    The `deptree_fill` procedure populates the `deptree` and `ideptree` views to display a tabular representation of all dependent objects, direct and indirect. You pass the object type, object owner, and object name to the `deptree_fill` procedure.

36. What does the `dbms_output` package do?

    The `dbms_output` package enables you to send messages from stored procedures, packages, and triggers.

37. How do you write "This procedure works." from within a PL/SQL program by using `dbms_output`?

    `DBMS_OUTPUT.PUT_LINE('This procedure works.');`

38. What does `dbms_sql` do and how does this compare with Native Dynamic SQL?

    `dbms_sql` enables you to embed dynamic DML, DDL, and DCL statements within a PL/SQL program. Native dynamic SQL allows you to place dynamic SQL statements directly into PL/SQL blocks. Native dynamic SQL in PL/SQL is easier to use than `dbms_sql`, requires much less application code, and performs better.

Practice 2: Solutions

1. Determine the output of the following code snippet.

```
SET SERVEROUTPUT ON
BEGIN
  UPDATE orders SET order_status = order_status;
  FOR v_rec IN ( SELECT order_id FROM orders )
    LOOP
      IF SQL%ISOPEN THEN
        DBMS_OUTPUT.PUT_LINE('TRUE - ' || SQL%ROWCOUNT);
      ELSE
        DBMS_OUTPUT.PUT_LINE('FALSE - ' || SQL%ROWCOUNT);
      END IF;
    END LOOP;
END;
/
```

Execute the code from the `lab_02_01.sql` file. **It will show `FALSE - 105` for each row fetched.**

2. Modify the following snippet of code to make better use of the `FOR UPDATE` clause and improve the performance of the program.

```
DECLARE
  CURSOR cur_update
    IS SELECT * FROM customers
    WHERE credit_limit < 5000 FOR UPDATE;
BEGIN
  FOR v_rec IN cur_update
    LOOP
      IF v_rec IS NOT NULL THEN
        UPDATE customers
          SET credit_limit = credit_limit + 200
          WHERE customer_id = v_rec.customer_id;
      END IF;
    END LOOP;
END;
/
```

Practice 2: Solutions (continued)

Modify the file `lab_02_02.sql` file as shown below:

```
DECLARE
  CURSOR cur_update
  IS SELECT * FROM customers
  WHERE credit_limit < 5000 FOR UPDATE;
BEGIN
  FOR v_rec IN cur_update
    LOOP
      UPDATE customers
      SET credit_limit = credit_limit + 200
      WHERE CURRENT OF cur_update;
    END LOOP;
END;
/
```

Alternatively, you can execute the code from the `sol_02_02.sql` file.

1. Create a package specification that defines subtypes, which can be used for the `warranty_period` field of the product_information table. Name this package `MY_TYPES`. The type needs to hold the month and year for a warranty period.

```
CREATE OR REPLACE PACKAGE mytypes
IS
  TYPE typ_warranty
    IS RECORD (month POSITIVE, year PLS_INTEGER);
  SUBTYPE warranty IS typ_warranty; -- based on RECORD type
END mytypes;
/
```

4. Create a package named `SHOW_DETAILS` that contains two subroutines. The first subroutine should show order details for the given `order_id`. The second subroutine should show customer details for the given `customer_id`, including the customer Id, first name, phone numbers, credit limit, and email address.

Both the subroutines should use the cursor variable to return the necessary details.

```
CREATE OR REPLACE PACKAGE show_details AS

TYPE rt_order IS REF CURSOR RETURN orders%ROWTYPE;

TYPE typ_cust_rec IS RECORD
  (cust_id NUMBER(6), cust_name VARCHAR2(20),
   custphone customers.phone_numbers%TYPE,
   credit NUMBER(9,2), cust_email VARCHAR2(30));
TYPE rt_cust IS REF CURSOR RETURN typ_cust_rec;

PROCEDURE get_order(p_orderid IN NUMBER, p_cv_order IN OUT rt_order);
```

Practice 2: Solutions (continued)

```
PROCEDURE get_cust(p_custid IN NUMBER, p_cv_cust IN OUT rt_cust);
END show_details;
/

CREATE OR REPLACE PACKAGE BODY show_details AS
PROCEDURE get_order
  (p_orderid IN NUMBER, p_cv_order IN OUT rt_order)
IS
BEGIN
  OPEN p_cv_order FOR
    SELECT * FROM order
      WHERE order_id = p_orderid;
-- CLOSE p_cv_order
END;

PROCEDURE get_cust
  (p_custid IN NUMBER, p_cv_cust IN OUT rt_cust)
IS
BEGIN
  OPEN p_cv_cust FOR
    SELECT customer_id, cust_first_name, phone_numbers, credit_limit,
         cust_email FROM customers
      WHERE customer_id = p_custid;
-- CLOSE p_cv_cust
END;
END;
/
```

Alternatively, you can execute the code from the `sol_02_04.sql` file.

Practice 3: Solutions

Collection Analysis

1. Examine the following definitions. Run the `lab_03_01.sql` script to create these objects.

```
CREATE TYPE typ_item AS OBJECT  --create object
 (prodid  NUMBER(5),
  price   NUMBER(7,2) )
/
CREATE TYPE typ_item_nst -- define nested table type
  AS TABLE OF typ_item
/
CREATE TABLE POrder (  -- create database table
    ordid NUMBER(5),
    supplier   NUMBER(5),
    requester  NUMBER(4),
    ordered    DATE,
    items typ_item_nst)
    NESTED TABLE items STORE AS item_stor_tab
/
@lab_03_01
```

2. The code shown below generates an error. Run the `lab_03_02.sql` script to generate and view the error.

```
BEGIN
  -- Insert an order
  INSERT INTO pOrder
    (ordid, supplier, requester, ordered, items)
    VALUES (1000, 12345, 9876, SYSDATE, NULL);
  -- insert the items for the order created
  INSERT INTO TABLE (SELECT items
                FROM   pOrder
                WHERE  ordid = 1000)
   VALUES(typ_item(99, 129.00));
END;
/
@lab_03_02
```

Why is the error occurring?

The error: ORA-22908: reference to NULL table value is resulting from setting the table columns to NULL.

Practice 3: Solutions (continued)

How can you fix the error?

Always use a nested table's default constructor to initialize it:

```
TRUNCATE TABLE pOrder;

-- A better approach is to avoid setting the table
-- column to NULL, and instead, use a nested table's
-- default constructor to initialize
BEGIN
  -- Insert an order
  INSERT INTO pOrder
    (ordid, supplier, requester, ordered, items)
    VALUES (1000, 12345, 9876, SYSDATE,
            typ_item_nst(typ_item(99, 129.00)));
END;
/

-- However, if the nested table is set to NULL, you can
-- use an UPDATE statement to set its value.
BEGIN
  -- Insert an order
  INSERT INTO pOrder
    (ordid, supplier, requester, ordered, items)
    VALUES (1000, 12345, 9876, SYSDATE, null);
  -- Once the nested table is set to null, use the update
  -- update statement
  UPDATE pOrder
    SET items = typ_item_nst(typ_item(99, 129.00))
    WHERE ordid = 1000
END;
/
```

Practice 3: Solutions (continued)

3. Examine the following code. This code produces an error. Which line causes the error, and how do you fix it? (**Note:** You can run the `lab_03_03.sql` script to view the error output).

```
DECLARE
  TYPE credit_card_typ
  IS VARRAY(100) OF VARCHAR2(30);
  v_mc   credit_card_typ := credit_card_typ();
  v_visa credit_card_typ := credit_card_typ();
  v_am   credit_card_typ;
  v_disc credit_card_typ := credit_card_typ();
  v_dc   credit_card_typ := credit_card_typ();
BEGIN
  v_mc.EXTEND;
  v_visa.EXTEND;
  v_am.EXTEND;
  v_disc.EXTEND;
  v_dc.EXTEND;
END;
/
```

This causes an ORA-06531: Reference to uninitialized collection. To fix it, initialize the `v_am` variable by using the same technique as the others:

```
DECLARE
  TYPE credit_card_typ
  IS VARRAY(100) OF VARCHAR2(30);

  v_mc   credit_card_typ := credit_card_typ();
  v_visa credit_card_typ := credit_card_typ();
  v_am   credit_card_typ := credit_card_typ();
  v_disc credit_card_typ := credit_card_typ();
  v_dc   credit_card_typ := credit_card_typ();

BEGIN
  v_mc.EXTEND;
  v_visa.EXTEND;
  v_am.EXTEND;
  v_disc.EXTEND;
  v_dc.EXTEND;
END;
/
```

Practice 3: Solutions (continued)

In the following practice exercises, you will implement a nested table column in the
CUSTOMERS table and write PL/SQL code to manipulate the nested table.

4. Create a nested table to hold credit card information.

    Create an object type called `typ_cr_card`. It should have the following specification:

```
card_type  VARCHAR2(25)
card_num   NUMBER
```

    Create a nested table type called `typ_cr_card_nst` that is a table of `typ_cr_card`.

```
CREATE TYPE typ_cr_card AS OBJECT  --create object
 (card_type  VARCHAR2(25),
  card_num   NUMBER);
/
CREATE TYPE typ_cr_card_nst -- define nested table type
  AS TABLE OF typ_cr_card;
/
```

    Add a column to the CUSTOMERS table called `credit_cards`. Make this column a nested
    table of type `typ_cr_card_nst`. You can use the following syntax:

```
ALTER TABLE customers ADD
  credit_cards typ_cr_card_nst
   NESTED TABLE credit_cards STORE AS c_c_store_tab;
```

5. Create a PL/SQL package that manipulates the `credit_cards` column in the
   CUSTOMERS table.

   Open the `lab_03_05.sql` file. It contains the package specification and part of the
   package body. Complete the code so that the package:

   - Inserts credit card information (the credit card name and number for a specific
     customer.)

   - Displays credit card information in an unnested format.

```
CREATE OR REPLACE PACKAGE credit_card_pkg
IS
  PROCEDURE update_card_info
    (p_cust_id NUMBER, p_card_type VARCHAR2, p_card_no VARCHAR2);

  PROCEDURE display_card_info
    (p_cust_id NUMBER);
END credit_card_pkg;  -- package spec
/
```

Practice 3: Solutions (continued)

```
CREATE OR REPLACE PACKAGE BODY credit_card_pkg
IS
  PROCEDURE update_card_info
    (p_cust_id NUMBER, p_card_type VARCHAR2, p_card_no VARCHAR2)
  IS
    v_card_info typ_cr_card_nst;
    i INTEGER;
  BEGIN
    SELECT credit_cards
      INTO v_card_info
      FROM customers
      WHERE customer_id = p_cust_id;
    IF v_card_info.EXISTS(1) THEN  -- cards exist, add more
      i := v_card_info.LAST;
      v_card_info.EXTEND(1);
      v_card_info(i+1) := typ_cr_card(p_card_type, p_card_no);
      UPDATE customers
        SET  credit_cards = v_card_info
        WHERE customer_id = p_cust_id;
    ELSE    -- no cards for this customer yet, construct one
      UPDATE customers
        SET  credit_cards = typ_cr_card_nst
             (typ_cr_card(p_card_type, p_card_no))
        WHERE customer_id = p_cust_id;
    END IF;
  END update_card_info;

  PROCEDURE display_card_info
    (p_cust_id NUMBER)
  IS
    v_card_info typ_cr_card_nst;
    i INTEGER;
  BEGIN
    SELECT credit_cards
      INTO v_card_info
      FROM customers
      WHERE customer_id = p_cust_id;
    IF v_card_info.EXISTS(1) THEN
      FOR idx IN v_card_info.FIRST..v_card_info.LAST LOOP
      DBMS_OUTPUT.PUT('Card Type: ' || v_card_info(idx).card_type || ' ');
      DBMS_OUTPUT.PUT_LINE('/ Card No: ' || v_card_info(idx).card_num );
      END LOOP;
    ELSE
      DBMS_OUTPUT.PUT_LINE('Customer has no credit cards.');
    END IF;
  END display_card_info;
END credit_card_pkg;  -- package body
/
```

Practice 3: Solutions (continued)

6. Test your package with the following statements and output:

```
SET SERVEROUT ON

EXECUTE credit_card_pkg.display_card_info(120)
Customer has no credit cards.
PL/SQL procedure successfully completed.

EXECUTE credit_card_pkg.update_card_info -
    (120, 'Visa', 11111111)
PL/SQL procedure successfully completed.


SELECT credit_cards
FROM   customers
WHERE  customer_id = 120;

CREDIT_CARDS(CARD_TYPE, CARD_NUM)
--------------------------------------------------
TYP_CR_CARD_NST(TYP_CR_CARD('Visa', 11111111))

EXECUTE credit_card_pkg.display_card_info(120)
Card Type: Visa / Card No: 11111111
PL/SQL procedure successfully completed.

EXECUTE credit_card_pkg.update_card_info -
    (120, 'MC', 2323232323)
PL/SQL procedure successfully completed.

EXECUTE credit_card_pkg.update_card_info -
    (120, 'DC', 4444444)
PL/SQL procedure successfully completed.


EXECUTE credit_card_pkg.display_card_info(120)
Card Type: Visa / Card No: 11111111
Card Type: MC / Card No: 2323232323
Card Type: DC / Card No: 4444444
PL/SQL procedure successfully completed.
```

Practice 3: Solutions (continued)

7. Write a `SELECT` statement against the `CREDIT_CARDS` column to unnest the data. Use the `TABLE` expression.

    For example, if the `SELECT` statement returns:

```
SELECT credit_cards
FROM   customers
WHERE  customer_id = 120;


CREDIT_CARDS(CARD_TYPE, CARD_NUM)
----------------------------------------------------------
TYP_CR_CARD_NST(TYP_CR_CARD('Visa', 11111111), TYP_CR_CARD('MC',
2323232323), TYP_CR_CARD('DC', 4444444))

-- Use the table expression so that the result is:

CUSTOMER_ID CUST_LAST_NAME  CARD_TYPE      CARD_NUM
----------- --------------- ------------- -----------
        120 Higgins         Visa             11111111
        120 Higgins         MC             2323232323
        120 Higgins         DC                4444444
```

```
SELECT c1.customer_id, c1.cust_last_name, c2.*
FROM   customers c1, TABLE(c1.credit_cards) c2
WHERE  customer_id = 120;
```

Practice 4: Solutions

1. An external C routine definition is created for you. The `.c` file is stored in the `$HOME/labs` directory on the database server. This function returns the tax amount based on the total sales figure passed to it as a parameter. The name of the `.c` file is named as `calc_tax.c`. The shared object filename is `calc_tax.so`. The function is defined as:

```
calc_tax(n)
  int n;
  {
     int tax;
     tax=(n*8)/100;
     return(tax);
  }
```

   a. Create a `calc_tax.so` file using the following command:

```
cc -shared -o calc_tax.so calc_tax.c
```

   b. Copy the file `calc_tax.so` to `$ORACLE_HOME/bin` directory using the following command:

```
cp calc_tax.so $ORACLE_HOME/bin
```

   c. Log in to SQL*Plus. Create the library object. Name the library object `c_code` and define its path as:

```
CREATE OR REPLACE LIBRARY c_code AS '$ORACLE_HOME/bin/calc_tax.so';
```

   d. Create a function named `call_c` to publish the external C routine. This fuction has one numeric parameter and it returns a binary integer. Identify the `AS LANGUAGE`, `LIBRARY`, and `NAME` clauses of the function.

```
CREATE OR REPLACE FUNCTION call_c
(x BINARY_INTEGER)
RETURN BINARY_INTEGER
AS LANGUAGE C
LIBRARY c_code
NAME "calc_tax";
/
```

Practice 4: Solutions (continued)

e. Create a procedure to call the `call_c` function created in the the previous step. Name this procedure `c_output`. It has one numeric parameter. Include a `DBMS_OUTPUT.PUT_LINE` statement so that you can view the results returned from your C function.

```
CREATE OR REPLACE PROCEDURE c_output
  (p_in IN BINARY_INTEGER)
IS
  i BINARY_INTEGER;
BEGIN
  i := call_c(p_in);
  DBMS_OUTPUT.PUT_LINE('The total tax is: ' || i);
END c_output;
/
```

f. Set the serveroutput ON.

```
SET SERVEROUTPUT ON
```

g. Execute the `c_output` procedure.

```
EXECUTE c_output(1000000)
The total tax is: 8000

PL/SQL procedure successfully completed.
```

Practice 4: Solutions (continued)

2. A Java method definition is created for you. The method accepts a 16-digit credit card number as the argument and returns the formatted credit card number (4 digits followed by a space). The `.java` file is stored in your `$HOME/labs` directory. The name of the .class file is `FormatCreditCardNo.class`. The method is defined as:

```
public class FormatCreditCardNo
{
  public static final void formatCard(String[] cardno)
  {
  int count=0, space=0;
  String oldcc=cardno[0];
  String[] newcc= {""};
  while (count<16)
  {
  newcc[0]+= oldcc.charAt(count);
  space++;
  if (space ==4)
  {  newcc[0]+=" "; space=0;   }
  count++;
  }
  cardno[0]=newcc [0];
  }
}
```

a. Load the `.java` source file. From the operating system, type:

```
loadjava –user oe/oe FormatCreditCardNo.java
```

b. Publish the Java class method by defining a PL/SQL procedure named `CCFORMAT`. This procedure accepts one `IN OUT` parameter.

Use the following definition for the `NAME` parameter:

`NAME 'FormatCreditCardNo.formatCard(java.lang.String[])';`

Create a file named `ccformat.sql` and enter the following code.

```
CREATE OR REPLACE PROCEDURE ccformat
(x IN OUT VARCHAR2)
AS LANGUAGE JAVA
NAME 'FormatCreditCardNo.formatCard(java.lang.String[])';
/
```

Save `ccformat.sql`.

Practice 4: Solutions (continued)

   c. Execute the Java class method. Define a SQL*Plus variable, intialize it, run the `ccformat.sql` file and use the execute command to execute the `ccformat` procedure. Finally, print the SQL*Plus variable.

```
VARIABLE x VARCHAR2(20)
EXECUTE :x := '1234567887654321'
PL/SQL procedure successfully completed.

@ccformat
Procedure created.

EXECUTE ccformat(:x)
PL/SQL procedure successfully completed.

PRINT x
X
----------------------------------------
1234 5678 8765 4321
```

Practice 5: Solutions

1. Create a PL/SQL server page to display order information. The name of the procedure that you are creating is `show_orders`.

    a. Open the `lab_05_01.psp` file. This file contains some HTML code.

    b. At the top of the file, include these directives:

```
<%@ page language="PL/SQL" %>
<%@ plsql procedure="show_orders" %>
```

    c. Use the following SQL statement to retrieve the order details. Place the following statement in the `FOR` loop:

```
SELECT order_id, order_mode, customer_id, order_status,
       order_total, call_c(order_total) tax, sales_rep_id
FROM   orders
ORDER BY order_id;
```
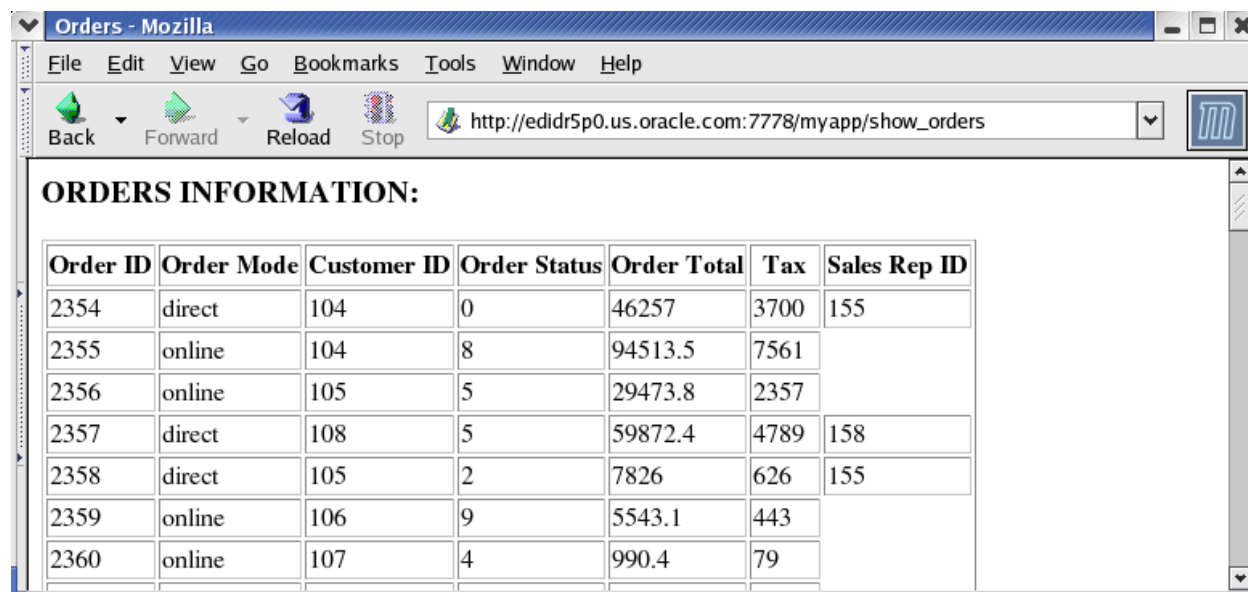
    d. Load the PSP file from `$HOME/labs`, type:

```
loadpsp -replace -user oe/oe lab_05_01.psp
```

Note: If the HTTP Server is not started, please start it using the following command:

```
opmnctl startall
```

    e. From your browser, request the `show_orders` PSP as shown below.

Open `sol_05_01.psp` to see the modified code.

Practice 5: Solutions (continued)

2. Create a PL/SQL server page to display the following customer information:

```
CUSTOMER_ID
CUST_FIRST_NAME
CUST_LAST_NAME
CREDIT_LIMIT
CUST_EMAIL
```

The name of the procedure is SHOW_CUST and you need to pass the CUSTOMER_ID as the parameter.

a. Open the lab_05_02a.psp file. This file contains some HTML code.

b. At the top of the file, include these directives:

```
<%@ page language="PL/SQL" %>
<%@ plsql procedure="show_cust" %>
<%@ plsql parameter="custid" %> type="NUMBER" default="101" %>
```

c. Place the parameter as shown in the following command:

```
<p>Following are the details for the Customer ID <%= custid %>
```

d. Use the following SQL statement to retrieve customer information. Place this statement in the FOR loop within the lab_05_02a.psp file.

```
SELECT * FROM customers WHERE customer_id = custid;
```

Note: You can access the sol_05_02a.psp file for the modified code.

e. Load the PSP file from $HOME/labs, type:

```
loadpsp -replace -user oe/oe lab_05_02a.psp
```

f. From the browser, request the show_cust PSP. By default it will show details for CUSTOMER_ID 101 because that is the specified default value.
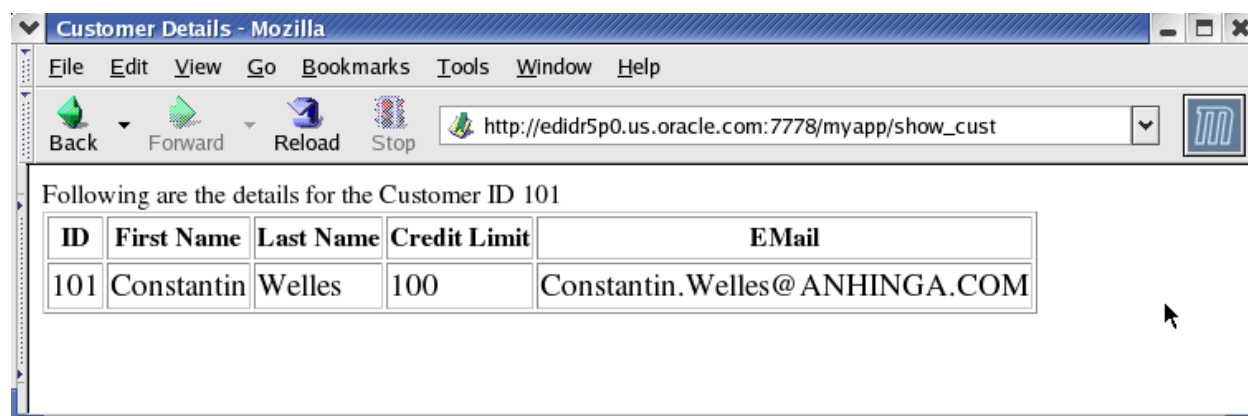
Practice 5: Solutions (continued)

      g. To see details for other customers, pass the parameter as shown below.
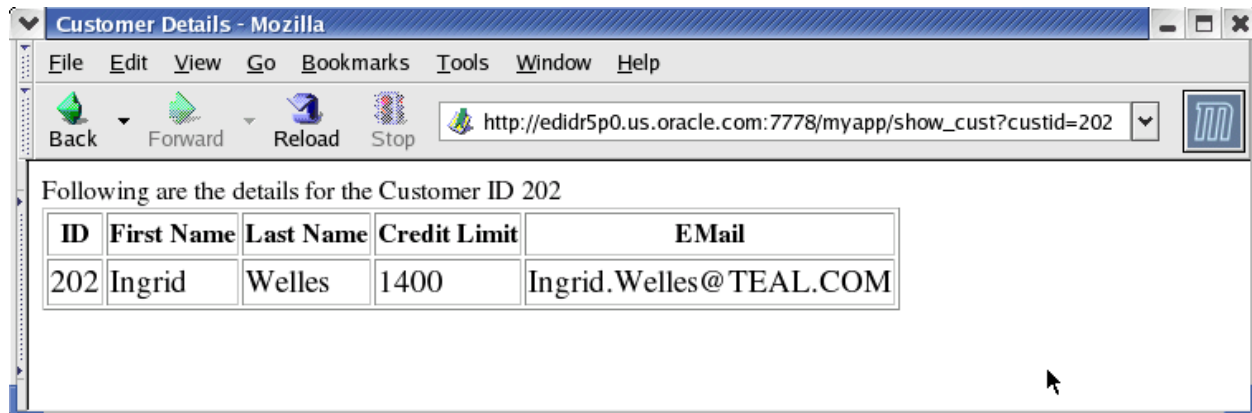
      h. To create an HTML form for calling the PSP, open `lab_05_02b.psp` and add the highlighted details.

```
<%@ page language="PL/SQL" %>
<%@ plsql procedure="show_cust_call" %>
<%@ plsql parameter="custid" %> type="NUMBER" default="101" %>
<HTML>
<BODY>
<form method="POST" action="show_cust">
<p>Enter the Customer ID:
<input type="text" name="custid">
<input type="submit" value="Submit">
</form>
</BODY>
</HTML>
```
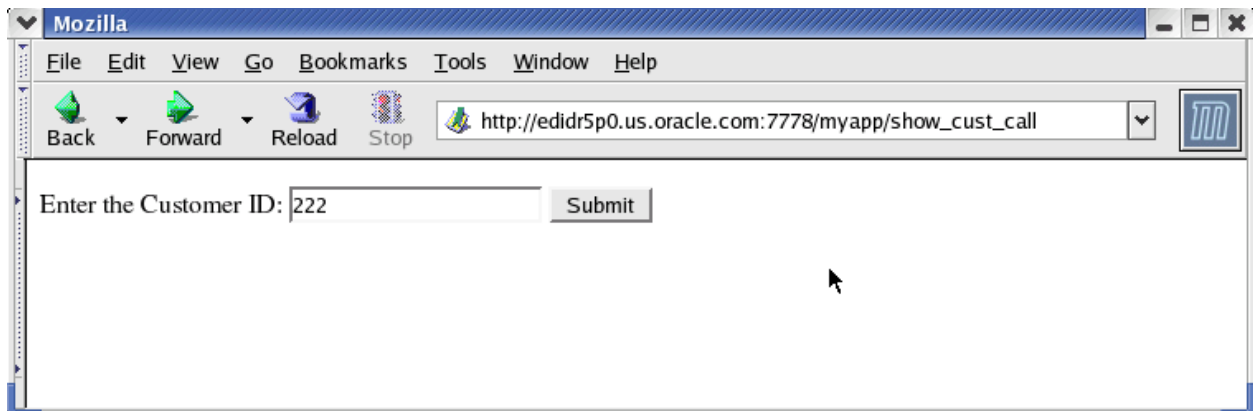
      i. Save the PSP file.

Note: You can access the `sol_05_02b.psp` file for the modified code.

      j. Load the PSP file from `$HOME/labs`, and enter:

```
loadpsp -replace -user oe/oe lab_05_02b.psp
```

Practice 5: Solutions (continued)

     k. From the browser, request `show_cust_call` PSP. Enter the Customer ID and click the Submit button.

     l. Note that the form in turn calls the `show_cust` PSP and then displays the details.
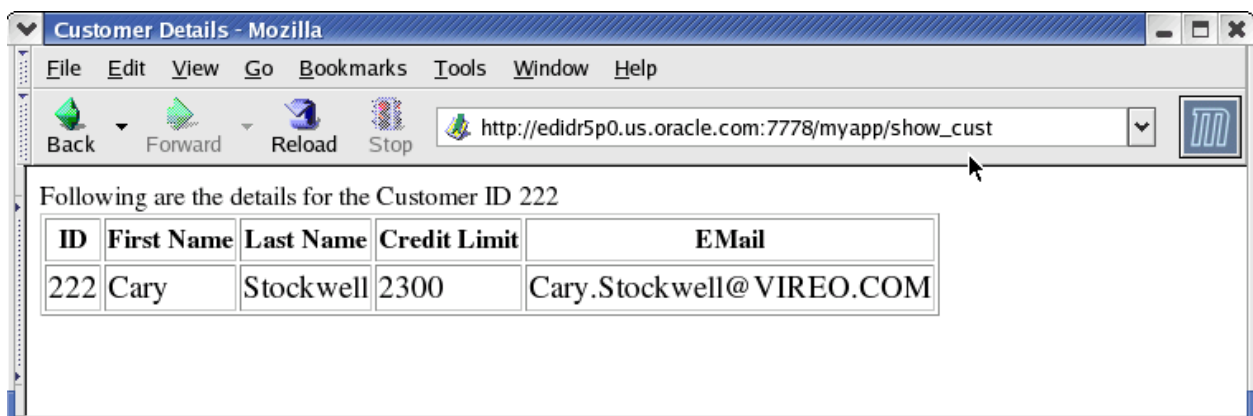
Practice 6: Solutions

In this practice you will define an application context and security policy to implement the policy: "Sales Representatives can see their own order information only in the ORDERS table." You will create sales representative IDs to test the success of your implementation.

Examine the definition of the ORDERS table, and the sales representative's data:

1. Examine, then run the lab_06_01.sql script.

    This script will create the sales representative's ID accounts with appropriate privileges to access the database:

```
CONNECT /AS sysdba

CREATE USER sr153 IDENTIFIED BY oracle
 DEFAULT TABLESPACE USERS
 TEMPORARY TABLESPACE TEMP
 QUOTA UNLIMITED ON USERS;

CREATE USER sr154 IDENTIFIED BY oracle
 DEFAULT TABLESPACE USERS
 TEMPORARY TABLESPACE TEMP
 QUOTA UNLIMITED ON USERS;

...

CREATE USER sr163 IDENTIFIED BY oracle
 DEFAULT TABLESPACE USERS
 TEMPORARY TABLESPACE TEMP
 QUOTA UNLIMITED ON USERS;

GRANT create session
    , alter session
   TO sr153, sr154, sr155, sr156, sr158, sr159,
   sr160, sr161, sr163;
GRANT SELECT, INSERT, UPDATE, DELETE ON
  oe.orders TO sr153, sr154, sr155, sr156, sr158,
                sr159, sr160, sr161, sr163;
GRANT SELECT, INSERT, UPDATE, DELETE ON
  oe.order_items TO sr153, sr154, sr155, sr156, sr158,
                sr159, sr160, sr161, sr163;

CREATE PUBLIC SYNONYM orders FOR oe.orders;
CREATE PUBLIC SYNONYM orders FOR oe.order_items;

CONNECT oe/oe
```

```
@lab_06_01.sql
```

Practice 6: Solutions (continued)

2. Set up an application context:

   Connect to the database as SYSDBA before creating this context.

   Create an application context named sales_orders_ctx.

   Associate this context to the oe.sales_orders_pkg.

```
CONNECT /AS sysdba

CREATE CONTEXT sales_orders_ctx
USING oe.sales_orders_pkg;
```

3. Connect as OE/OE.

   Examine this package specification:

```
CREATE OR REPLACE PACKAGE sales_orders_pkg
IS
 PROCEDURE set_app_context;
 FUNCTION the_predicate
  (p_schema VARCHAR2, p_name VARCHAR2)
   RETURN VARCHAR2;
END sales_orders_pkg;    -- package spec
/
```

   Create this package specification and then the package body in the OE schema.

   When you create the package body, set up two constants as follows:

```
c_context CONSTANT VARCHAR2(30) := 'SALES_ORDER_CTX';
c_attrib  CONSTANT VARCHAR2(30) := 'SALES_REP';
```

   Use these constants in the SET_APP_CONTEXT procedure to set the application context
   to the current user.

Practice 6: Solutions (continued)

```
CREATE OR REPLACE PACKAGE BODY sales_orders_pkg
IS
  c_context CONSTANT VARCHAR2(30) := 'SALES_ORDERS_CTX';
  c_attrib  CONSTANT VARCHAR2(30) := 'SALES_REP';

PROCEDURE set_app_context
  IS
    v_user VARCHAR2(30);
BEGIN
  SELECT user INTO v_user FROM dual;
  DBMS_SESSION.SET_CONTEXT
    (c_context, c_attrib, v_user);
END set_app_context;

FUNCTION the_predicate
(p_schema VARCHAR2, p_name VARCHAR2)
RETURN VARCHAR2
IS
  v_context_value VARCHAR2(100) :=
    SYS_CONTEXT(c_context, c_attrib);
  v_restriction VARCHAR2(2000);
BEGIN
  IF v_context_value LIKE 'SR%'  THEN
    v_restriction :=
     'SALES_REP_ID =
      SUBSTR(''' || v_context_value || ''', 3, 3)';
  ELSE
    v_restriction := null;
  END IF;
  RETURN v_restriction;
END the_predicate;

END sales_orders_pkg; -- package body
/
```

Practice 6: Solutions (continued)

4. Connect as SYSDBA and define the policy.

   Use DBMS_RLS.ADD_POLICY to define the policy.

   Use these specifications for the parameter values:

   ```
   object_schema    OE
   object_name      ORDERS
   policy_name      OE_ORDERS_ACCESS_POLICY
   function_schema  OE
   policy_function  SALES_ORDERS_PKG.THE_PREDICATE
   statement_types  SELECT, INSERT, UPDATE, DELETE
   update_check     FALSE,
   enable           TRUE);
   ```

```
CONNECT /as sysdba

DECLARE
BEGIN
  DBMS_RLS.ADD_POLICY (
    'OE',
    'ORDERS',
    'OE_ORDERS_ACCESS_POLICY',
    'OE',
    'SALES_ORDERS_PKG.THE_PREDICATE',
    'SELECT, INSERT, UPDATE, DELETE',
    FALSE,
    TRUE);
END;
/
```

5. Connect as SYSDBA and create a logon trigger to implement fine-grained access control. You can call the trigger SET_ID_ON_LOGON. This trigger causes the context to be set as each user is logged on.

```
CONNECT /as sysdba

CREATE OR REPLACE TRIGGER set_id_on_logon
AFTER logon on DATABASE
BEGIN
  oe.sales_orders_pkg.set_app_context;
END;
/
```

Practice 6: Solutions (continued)

6. Test the fine-grained access implementation. Connect as your SR user and query the ORDERS table. For example, your results should match:

```
CONNECT sr153/oracle

SELECT sales_rep_id, COUNT(*)
FROM   orders
GROUP BY sales_rep_id;

SALES_REP_ID   COUNT(*)
------------ ----------
         153          5

CONNECT sr154/oracle

SELECT sales_rep_id, COUNT(*)
FROM   orders
GROUP BY sales_rep_id;

SALES_REP_ID   COUNT(*)
------------ ----------
         154         10
```

Note

During debugging, you may need to disable or remove some of the objects created for this lesson.

If you need to disable the logon trigger, issue the command:

```
ALTER TRIGGER set_id_on_logon DISABLE;
```

If you need to remove the policy you created, issue the command:

```
EXECUTE DBMS_RLS.DROP_POLICY('OE', 'ORDERS',-
   'OE_ORDERS_ACCESS_POLICY')
```

Practice 7: Solutions

1. In this exercise, you will pin the fine-grained access package created in Lesson 6.

    Note: If you have not completed practice 6, run the following files in the $HOME/soln folder:

```
@sol_06_02.sql
@sol_06_03.sql
@sol_06_04.sql
@sol_06_05.sql
```

Using the DBMS_SHARED_POOL.KEEP procedure, pin your SALES_ORDERS_PKG.

```
EXECUTE sys.dbms_shared_pool.keep('SALES_ORDERS_PKG')
```

Execute the DBMS_SHARED_POOL.SIZES procedure to see the objects in the shared pool that are larger than 500 kilobytes.

```
SET SERVEROUTPUT ON
EXECUTE sys.dbms_shared_pool.sizes(500)
```

2. Open the lab_07_02.sql file and examine the package (the package body is shown below):

```
CREATE OR REPLACE PACKAGE BODY credit_card_pkg
IS
  PROCEDURE update_card_info
    (p_cust_id NUMBER, p_card_type VARCHAR2, p_card_no VARCHAR2)
  IS
    v_card_info typ_cr_card_nst;
    i INTEGER;
  BEGIN
    SELECT credit_cards
      INTO v_card_info
      FROM customers
      WHERE customer_id = p_cust_id;
    IF v_card_info.EXISTS(1) THEN  -- cards exist, add more
      i := v_card_info.LAST;
      v_card_info.EXTEND(1);
      v_card_info(i+1) := typ_cr_card(p_card_type, p_card_no);
      UPDATE customers
        SET  credit_cards = v_card_info
        WHERE customer_id = p_cust_id;
    ELSE   -- no cards for this customer yet, construct one
      UPDATE customers
        SET  credit_cards = typ_cr_card_nst
           (typ_cr_card(p_card_type, p_card_no))
        WHERE customer_id = p_cust_id;
    END IF;
  END update_card_info;
-- continued on next page
```

```
-- continued from previous page.
  PROCEDURE display_card_info
    (p_cust_id NUMBER)
  IS
    v_card_info typ_cr_card_nst;
    i INTEGER;
  BEGIN
    SELECT credit_cards
      INTO v_card_info
      FROM customers
      WHERE customer_id = p_cust_id;
    IF v_card_info.EXISTS(1) THEN
      FOR idx IN v_card_info.FIRST..v_card_info.LAST LOOP
          DBMS_OUTPUT.PUT('Card Type: ' || v_card_info(idx).card_type
          || ' ');
        DBMS_OUTPUT.PUT_LINE('/ Card No: ' ||
          v_card_info(idx).card_num );
      END LOOP;
    ELSE
      DBMS_OUTPUT.PUT_LINE('Customer has no credit cards.');
    END IF;
  END display_card_info;
END credit_card_pkg;  -- package body
/
```

This code needs to be improved. The following issues exist in the code:

- The local variables use the INTEGER data type.

- The same SELECT statement is run in the two procedures.

- The same IF v_card_info.EXISTS(1) THEN statement is in the two procedures.

Practice 7: Solutions (continued)

3. To improve the code, make the following modifications:

Change the local INTEGER variables to use a more efficient data type.

Move the duplicated code into a function. The package specification for the modification is:

```
CREATE OR REPLACE PACKAGE credit_card_pkg
IS
  FUNCTION cust_card_info
    (p_cust_id NUMBER, p_card_info IN OUT typ_cr_card_nst )
    RETURN BOOLEAN;
  PROCEDURE update_card_info
    (p_cust_id NUMBER, p_card_type VARCHAR2, p_card_no VARCHAR2);
   PROCEDURE display_card_info
    (p_cust_id NUMBER);
END credit_card_pkg;  -- package spec
/
```

Have the function return TRUE if the customer has credit cards. The function should return FALSE if the customer does not have credit cards. Pass into the function an uninitialized nested table. The function places the credit card information into this uninitialized parameter.

```
CREATE OR REPLACE PACKAGE credit_card_pkg
IS
  FUNCTION cust_card_info
    (p_cust_id NUMBER, p_card_info IN OUT typ_cr_card_nst )
    RETURN BOOLEAN;

  PROCEDURE update_card_info
    (p_cust_id NUMBER, p_card_type VARCHAR2, p_card_no VARCHAR2);

  PROCEDURE display_card_info
    (p_cust_id NUMBER);

END credit_card_pkg;  -- package spec
/

-- continued on next page
```

Practice 7: Solutions (continued)

```
CREATE OR REPLACE PACKAGE BODY credit_card_pkg
IS
  FUNCTION cust_card_info
    (p_cust_id NUMBER, p_card_info IN OUT typ_cr_card_nst )
    RETURN BOOLEAN
  IS
    v_card_info_exists BOOLEAN;
  BEGIN
    SELECT credit_cards
      INTO p_card_info
      FROM customers
      WHERE customer_id = p_cust_id;
    IF p_card_info.EXISTS(1) THEN
      v_card_info_exists := TRUE;
    ELSE
      v_card_info_exists := FALSE;
    END IF;
    RETURN v_card_info_exists;
  END cust_card_info;

  PROCEDURE update_card_info
    (p_cust_id NUMBER, p_card_type VARCHAR2, p_card_no VARCHAR2)
  IS
    v_card_info typ_cr_card_nst;
    i PLS_INTEGER;
  BEGIN
    IF cust_card_info(p_cust_id, v_card_info) THEN
      -- cards exist, add more
      i := v_card_info.LAST;
      v_card_info.EXTEND(1);
      v_card_info(i+1) := typ_cr_card(p_card_type, p_card_no);
      UPDATE customers
        SET  credit_cards = v_card_info
        WHERE customer_id = p_cust_id;
    ELSE   -- no cards for this customer yet, construct one
      UPDATE customers
        SET  credit_cards = typ_cr_card_nst
            (typ_cr_card(p_card_type, p_card_no))
        WHERE customer_id = p_cust_id;
    END IF;
  END update_card_info;

-- continued on next page
```

Practice 7: Solutions (continued)

```
PROCEDURE display_card_info
    (p_cust_id NUMBER)
  IS
    v_card_info typ_cr_card_nst;
    i PLS_INTEGER;
  BEGIN
    IF cust_card_info(p_cust_id, v_card_info) THEN
      FOR idx IN v_card_info.FIRST..v_card_info.LAST LOOP
        DBMS_OUTPUT.PUT('Card Type: ' ||
        v_card_info(idx).card_type || ' ');
      DBMS_OUTPUT.PUT_LINE('/ Card No: ' ||
        v_card_info(idx).card_num );
      END LOOP;
    ELSE
      DBMS_OUTPUT.PUT_LINE('Customer has no credit cards.');
    END IF;
  END display_card_info;
END credit_card_pkg;  -- package body
/
```

4. Test your modified code with the following data:

```
EXECUTE credit_card_pkg.update_card_info -
       (120, 'AM EX', 55555555555)
PL/SQL procedure successfully completed.

EXECUTE credit_card_pkg.display_card_info(120)
Card Type: Visa / Card No: 11111111
Card Type: MC / Card No: 2323232323
Card Type: DC / Card No: 4444444
Card Type: AM EX / Card No: 55555555555

PL/SQL procedure successfully completed.
-- Note: If you did not complete Practice 3, your results
-- will be:
EXECUTE credit_card_pkg.display_card_info(120)
Card Type: AM EX / Card No: 55555555555

PL/SQL procedure successfully completed.
```

Practice 7: Solutions (continued)

5. Open the `lab_07_05a.sql` file. It contains the modified code from the previous question #3.

    You need to modify the UPDATE_CARD_INFO procedure to return information (using the RETURNING clause) about the credit cards being updated. Assume that this information will be used by another application developer in your team, who is writing a graphical reporting utility on customer credit cards, after a customer's credit card information is changed.

    Modify the code to use the RETURNING clause to find information about the row affected by the UPDATE statements.

```
CREATE OR REPLACE PACKAGE credit_card_pkg
IS
  FUNCTION cust_card_info
    (p_cust_id NUMBER, p_card_info IN OUT typ_cr_card_nst )
    RETURN BOOLEAN;

  PROCEDURE update_card_info
    (p_cust_id NUMBER, p_card_type VARCHAR2,
     p_card_no VARCHAR2, o_card_info OUT typ_cr_card_nst);

  PROCEDURE display_card_info
    (p_cust_id NUMBER);

END credit_card_pkg;  -- package spec
/
```

Practice 7: Solutions (continued)

```
... only the update_card_info procedure is changed in the body

PROCEDURE update_card_info
    (p_cust_id NUMBER, p_card_type VARCHAR2,
     p_card_no VARCHAR2, o_card_info OUT typ_cr_card_nst)
  IS
    v_card_info typ_cr_card_nst;
    i PLS_INTEGER;
  BEGIN
    IF cust_card_info(p_cust_id, v_card_info) THEN
      -- cards exist, add more
      i := v_card_info.LAST;
      v_card_info.EXTEND(1);
      v_card_info(i+1) := typ_cr_card(p_card_type, p_card_no);
      UPDATE customers
        SET  credit_cards = v_card_info
        WHERE customer_id = p_cust_id
        RETURNING credit_cards INTO o_card_info;
    ELSE    -- no cards for this customer yet, construct one
      UPDATE customers
        SET  credit_cards = typ_cr_card_nst
            (typ_cr_card(p_card_type, p_card_no))
        WHERE customer_id = p_cust_id
        RETURNING credit_cards INTO o_card_info;
    END IF;
END update_card_info;
...
```

You can test your modified code with the following procedure (contained in
`lab_07_05b.sql`):

```
CREATE OR REPLACE PROCEDURE test_credit_update_info
(p_cust_id NUMBER, p_card_type VARCHAR2, p_card_no NUMBER)
IS
  v_card_info typ_cr_card_nst;
BEGIN
  credit_card_pkg.update_card_info
    (p_cust_id, p_card_type, p_card_no, v_card_info);
END test_credit_update_info;
/
```

Practice 7: Solutions (continued)

Test your code with the following statements set in boldface:

```
EXECUTE test_credit_update_info(125, 'AM EX', 123456789)
PL/SQL procedure successfully completed.
SELECT credit_cards FROM customers WHERE customer_id = 125;
CREDIT_CARDS(CARD_TYPE, CARD_NUM)
------------------------------------------------------------
TYP_CR_CARD_NST(TYP_CR_CARD('AM EX', 123456789))
```

6. In this exercise, you will test exception handling with the SAVED EXCEPTIONS clause.

Run the lab_07_06a.sql file to create a test table:

```
CREATE TABLE card_table
(accepted_cards VARCHAR2(50) NOT NULL);
```

Open the lab_07_06b.sql file and run the contents:

```
DECLARE
  type typ_cards is table of VARCHAR2(50);
  v_cards typ_cards := typ_cards
  ( 'Citigroup Visa', 'Nationscard MasterCard',
    'Federal American Express', 'Citizens Visa',
    'International Discoverer', 'United Diners Club' );
BEGIN
  v_cards.Delete(3);
  v_cards.DELETE(6);
  FORALL j IN v_cards.first..v_cards.last
    SAVE EXCEPTIONS
    EXECUTE IMMEDIATE
   'insert into card_table (accepted_cards) values ( :the_card)'
    USING v_cards(j);
/
```

Note the output: This returns an "Error in Array DML (at line 11)," which is not very informational. The cause of this error is: one or more rows failed in the DML.

Practice 7: Solutions (continued)

6. (continued)

Open the `lab_07_06c.sql` file and run the contents:

```
DECLARE
  type typ_cards is table of VARCHAR2(50);
  v_cards typ_cards := typ_cards
  ( 'Citigroup Visa', 'Nationscard MasterCard',
    'Federal American Express', 'Citizens Visa',
    'International Discoverer', 'United Diners Club' );
  bulk_errors EXCEPTION;
  PRAGMA exception_init (bulk_errors, -24381 );
BEGIN
  v_cards.Delete(3);
  v_cards.DELETE(6);
  FORALL j IN v_cards.first..v_cards.last
    SAVE EXCEPTIONS
    EXECUTE IMMEDIATE
   'insert into card_table (accepted_cards) values ( :the_card)'
    USING v_cards(j);
 EXCEPTION
  WHEN  bulk_errors THEN
    FOR j IN 1..sql%bulk_exceptions.count
  LOOP
    Dbms_Output.Put_Line (
      TO_CHAR( sql%bulk_exceptions(j).error_index ) || ':
      ' || SQLERRM(-sql%bulk_exceptions(j).error_code) );
  END LOOP;
END;
/
```

Note the output:

ORA-22160: element at index [] does not exist

PL/SQL procedure successfully completed.

Why is the output different?

**The PL/SQL block raises the exception 22160 when it encounters an array element
that was deleted. The exception is handled and the block completes successfully.**

Practice 8: Solutions

In this exercise, you will profile the CREDIT_CARD_PKG package created in an earlier lesson.

1. Run the `lab_08_01.sql` script to create the CREDIT_CARD_PKG package.

```
@$HOME/labs/lab_08_01.sql
```

2. Run the `proftab.sql` script to create the profile tables under your schema.

```
@$HOME/labs/proftab.sql
```

3. Create a MY_PROFILER procedure to:

Start the profiler

Run the application

```
EXECUTE credit_card_pkg.update_card_info -
        (130, 'AM EX', 121212121212)
```

Flush the profiler data

Stop the profiler

```
CREATE OR REPLACE PROCEDURE my_profiler
(p_comment1 IN VARCHAR2, p_comment2 IN VARCHAR2)
IS
  v_return_code    NUMBER;
BEGIN
--start the profiler
  v_return_code:=DBMS_PROFILER.START_PROFILER
    (p_comment1, p_comment2);
  dbms_output.put_line
    ('Result from START: '||v_return_code);

-- now run a program...
  credit_card_pkg.update_card_info (130, 'AM EX', 121212121212);
--flush the collected data to the dictionary tables
  v_return_code := DBMS_PROFILER.FLUSH_DATA;
  dbms_output.put_line
    ('Result from FLUSH: '||v_return_code);
--stop profiling
  v_return_code := DBMS_PROFILER.STOP_PROFILER;
  dbms_output.put_line
    ('Result from STOP: '||v_return_code);
END;
/
```

Practice 8: Solutions (continued)

    4. Execute the MY_PROFILER procedure.

```
SET SERVEROUTPUT ON

EXECUTE my_profiler('Benchmark Run.' , 'This is the first run.')
```

    5. Analyze the results of profiling in the PLSQL_PROFILER tables.

```
SELECT runid, run_owner, run_date, run_comment,
       run_comment1, run_total_time
FROM   plsql_profiler_runs;

SELECT runid, unit_number, unit_type,
       unit_owner, unit_name
FROM   plsql_profiler_units inner
JOIN   plsql_profiler_runs
USING  ( runid );

SELECT line#, total_occur, total_time,
       min_time, max_time
FROM   plsql_profiler_data
WHERE  runid = 1 AND unit_number = 2;
```

In this exercise, you will trace the CREDIT_CARD_PKG package.

    6. Enable the CREDIT_CARD_PKG for tracing by using the ALTER statement with the COMPILE DEBUG option.

```
ALTER PACKAGE credit_card_pkg COMPILE DEBUG BODY;
```

    7. Start the trace session and trace all calls.

```
EXECUTE DBMS_TRACE.SET_PLSQL_TRACE(DBMS_TRACE.trace_all_calls)
```

    8. Run the credit_card_pkg.update_card_info procedure with the following data:

```
EXECUTE credit_card_pkg.update_card_info –
        (135, 'DC', 987654321)
```

Practice 8: Solutions (continued)

9. Disable tracing.

```
EXECUTE DBMS_TRACE.CLEAR_PLSQL_TRACE
```

10. Examine the trace information by querying the trace tables.

```
COLUMN event_comment    format a28
COLUMN event_proc_name format a18
COLUMN proc_name        format a17

SELECT proc_name, proc_line,
         event_proc_name, event_comment
FROM sys.plsql_trace_events
WHERE event_unit = 'CREDIT_CARD_PKG';
```

Table Descriptions

Schema Descriptions

Overall Description

The sample company portrayed by the Oracle Database Sample Schemas operates worldwide to fulfil orders for several different products. The company has several divisions:

- The Human Resources division tracks information about the employees and facilities of the company.
- The Order Entry division tracks product inventories and sales of the products of the company through various channels.
- The Sales History division tracks business statistics to facilitate business decisions.

Each of these divisions is represented by a schema. In this course, you have access to the objects in all of these schemas. However, the emphasis of the examples, demonstrations, and practices utilize the `Order Entry` (OE) schema.

All scripts necessary to create the sample schemas reside in the `$ORACLE_HOME/demo/schema/` folder.

Schema Descriptions (continued)

`Order Entry` (OE)

The company sells several categories of products, including computer hardware and software, music, clothing, and tools. The company maintains product information that includes product identification numbers, the category into which the product falls, the weight group (for shipping purposes), the warranty period if applicable, the supplier, the status of the product, a list price, a minimum price at which a product will be sold, and a URL for manufacturer information.

Inventory information is also recorded for all products, including the warehouse where the product is available and the quantity on hand. Because products are sold worldwide, the company maintains the names of the products and their descriptions in different languages.

The company maintains warehouses in several locations to facilitate filling customer orders. Each warehouse has a warehouse identification number, name, and location identification number.

Customer information is tracked in some detail. Each customer is assigned an identification number. Customer records include name, street address, city or province, country, phone numbers (up to five phone numbers for each customer), and postal code. Some customers order through the Internet, so e-mail addresses are also recorded. Because of language differences among customers, the company records the NLS language and territory of each customer. The company places a credit limit on its customers to limit the amount for which they can purchase at one time. Some customers have account managers, whom the company monitors. We keep track of a customer's phone number. These days, we never know how many phone numbers a customer might have, but we try to keep track of all of them. Because of the language differences of the customers, we identify the language and territory of each customer.

When a customer places an order, the company tracks the date of the order, the mode of the order, status, shipping mode, total amount of the order, and the sales representative who helped place the order. This may be the same individual as the account manager for a customer, it may be different, or, in the case of an order over the Internet, the sales representative is not recorded. In addition to the order information, we also track the number of items ordered, the unit price, and the products ordered.

For each country in which it does business, the company records the country name, currency symbol, currency name, and the region where the country resides geographically. This data is useful to interact with customers living in different geographic regions around the world.

`Order Entry(OE)`

OE

ORDER_ITEMS
order_id
line_item_id
product_id
unit_price
quantity

ORDERS
order_id
order_date
order_mode
customer_id
ship_mode
order_status
sales_rep_id
order_total

CUSTOMERS
customer_id
cust_first_name
cust_last_name
cust_address_typ
street_address
postal_code
city
state_province
country_id
phone_numbers
credit_limit
cust_email
account_mgr_id
nls_language
nls_territory

PRODUCT INFORMATION
product_id
category
weight_class
warranty_period
supplier_id
product_status
list_price
min_price
catalog_url

PRODUCT DESCRIPTIONS
product_id
language_id
product_name
product_description

INVENTORIES
product_id
warehouse_id
quantity_on_hand

WAREHOUSES
warehouse_id
warehouse_name
location_id
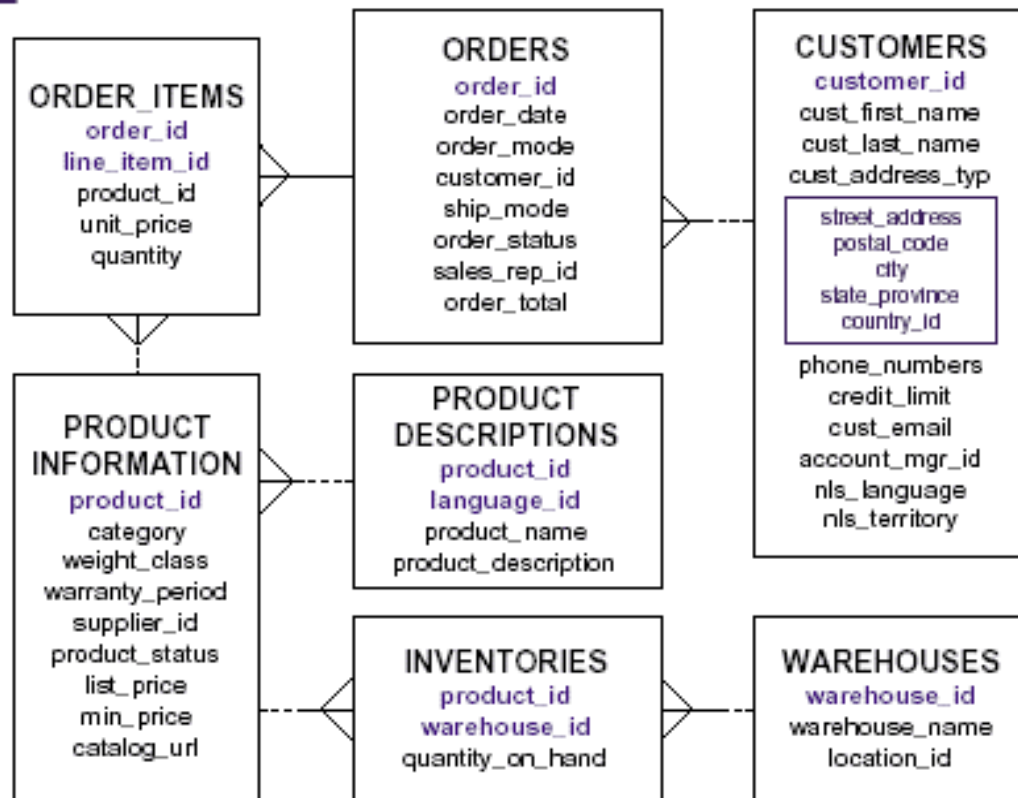
Schema Descriptions (continued)

Order Entry (OE) Row Counts

```
SELECT COUNT(*) FROM customers;
  COUNT(*)
----------
       319

SELECT COUNT(*) FROM inventories;
  COUNT(*)
----------
      1112

SELECT COUNT(*) FROM orders;
  COUNT(*)
----------
       105

SELECT COUNT(*) FROM order_items;
  COUNT(*)
----------
       665

SELECT COUNT(*) FROM product_descriptions;
  COUNT(*)
----------
      8640

SELECT COUNT(*) FROM product_information;
  COUNT(*)
----------
       288

SELECT COUNT(*) FROM warehouses;
  COUNT(*)
----------
         9
```

Human Resources (HR)

In the human resource records, each employee has an identification number, e-mail address, job identification code, salary, and manager. Some employees earn a commission in addition to their salary.

The company also tracks information about jobs within the organization. Each job has an identification code, job title, and a minimum and maximum salary range for the job. Some employees have been with the company for a long time and have held different positions within the company. When an employee switches jobs, the company records the start date and end date of the former job, the job identification number, and the department.

The sample company is regionally diverse, so it tracks the locations of not only its warehouses but also of its departments. Each company employee is assigned to a department. Each department is identified by a unique department number and a short name. Each department is associated with one location. Each location has a full address that includes the street address, postal code, city, state or province, and country code.

For each location where it has facilities, the company records the country name, currency symbol, currency name, and the region where the country resides geographically.

Oracle University and En-Sof Informatica E Treinamento Ltda use only
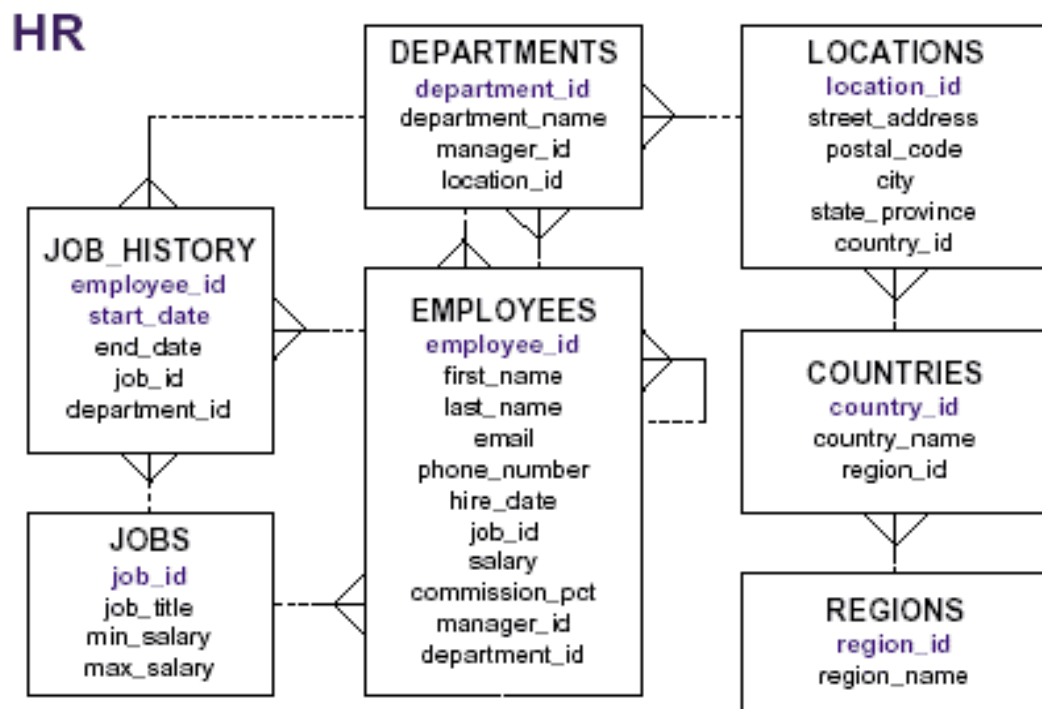
Schema Descriptions (continued)

Human Resources (HR) Row Counts

```
SELECT COUNT(*) FROM employees;
  COUNT(*)
----------
       107

SELECT COUNT(*) FROM departments;
  COUNT(*)
----------
        27

SELECT COUNT(*) FROM locations;
  COUNT(*)
----------
        23

SELECT COUNT(*) FROM countries;
  COUNT(*)
----------
        25

SELECT COUNT(*) FROM regions;
  COUNT(*)
----------
         4

SELECT COUNT(*) FROM jobs;
  COUNT(*)
----------
        19

SELECT COUNT(*) FROM job_history;
  COUNT(*)
----------
        10
```

C

Appendix C
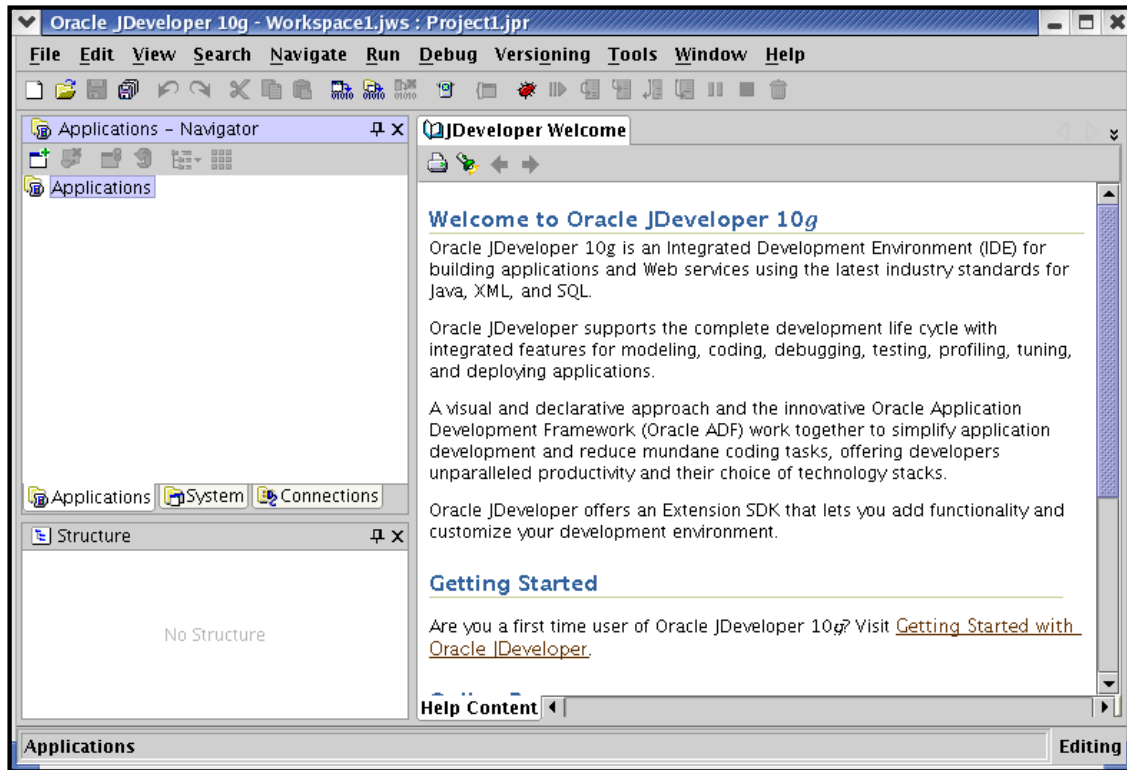
JDeveloper

JDeveloper

Oracle JDeveloper 10*g* is an integrated development environment (IDE) for developing and deploying Java applications and Web services. It supports every stage of the software development life cycle (SDLC) from modeling to deploying. It has the features to use the latest industry standards for Java, XML, and SQL while developing an application.

Oracle JDeveloper 10*g* initiates a new approach to J2EE development with the features that enables visual and declarative development. This innovative approach makes J2EE development simple and efficient.

Connection Navigator
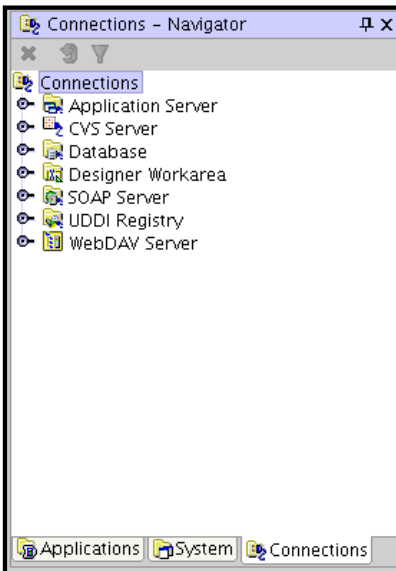
Connection Navigator

Using Oracle JDeveloper 10*g*, you can store the information necessary to connect to a database in an object called "connection." A connection is stored as part of the IDE settings, and can be exported and imported for easy sharing among groups of users. A connection serves several purposes from browsing the database and building applications, all the way through to deployment.

Application Navigator
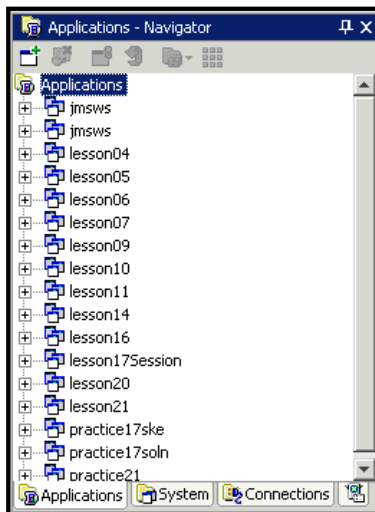
Application Navigator

The Application Navigator gives you a logical view of your application and the data it contains. The Application Navigator provides an infrastructure that the different extensions can plug into and use to organize their data and menus in a consistent, abstract manner. While the Application Navigator can contain individual files (such as Java source files), it is designed to consolidate complex data. Complex data types such as entity objects, UML diagrams, EJB, or Web services appear in this navigator as single nodes. The raw files that make up these abstract nodes appear in the Structure window.
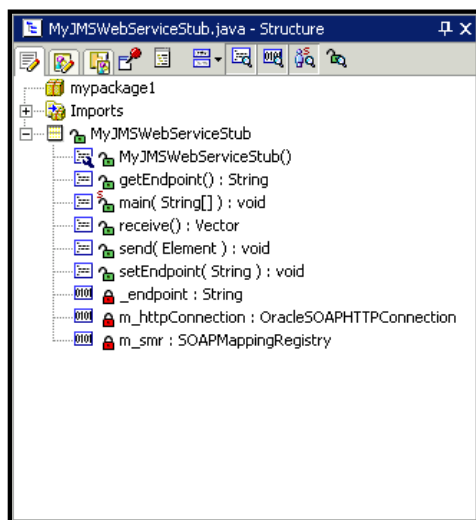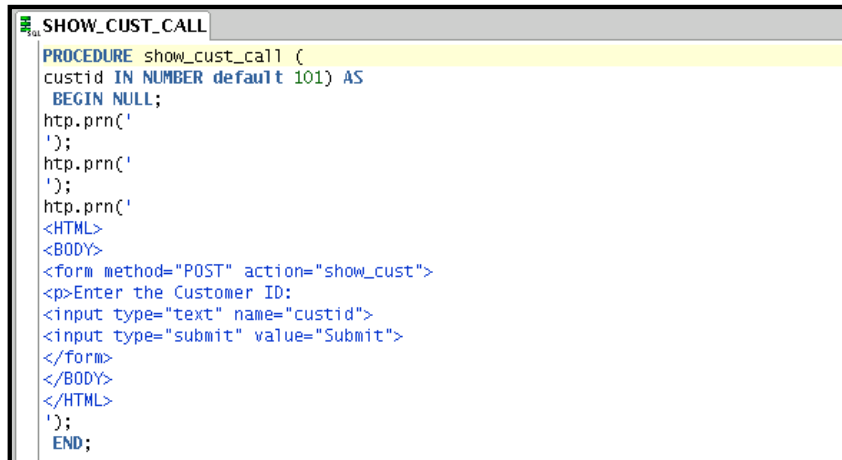
Structure Window

Structure Window

The Structure window offers a structural view of the data in the document currently selected in the active window of those windows that participate in providing structure: the navigators, the editors and viewers, and the Property Inspector.

In the Structure window, you can view the document data in a variety of ways. The structures available for display are based upon document type. For a Java file, you can view code structure, UI structure, or UI model data. For an XML file, you can view XML structure, design structure, or UI model data.

The Structure window is dynamic, tracking always the current selection of the active window (unless you freeze the window's contents on a particular view), as is pertinent to the currently active editor. When the current selection is a node in the navigator, the default editor is assumed. To change the view on the structure for the current selection, select a different structure tab.

Editor Window

Editor Window

You can view your project files all in one single editor window, you can open multiple views of the same file, or you can open multiple views of different files.

The tabs at the top of the editor window are the document tabs. Selecting a document tab gives that file focus, bringing it to the foreground of the window in the current editor.

The tabs at the bottom of the editor window for a given file are the editor tabs. Selecting an editor tab opens the file in that editor.

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.**
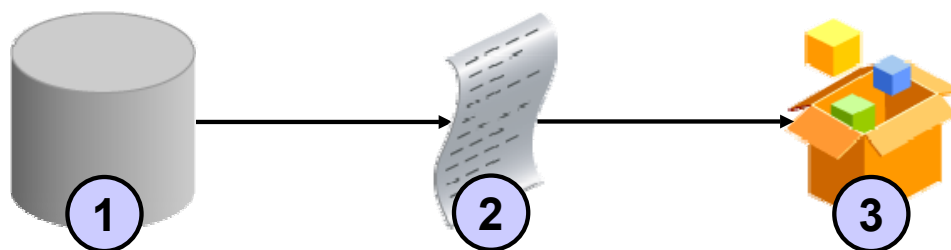
ORACLE

Deploying Java Stored Procedures

Create a deployment profile for Java stored procedures, then deploy the classes and, optionally, any public static methods in JDeveloper using the settings in the profile.

Deploying to the database uses the information provided in the Deployment Profile Wizard and two Oracle Database utilities:

- `loadjava` loads the Java class containing the stored procedures to an Oracle database.
- `publish` generates the PL/SQL call specific wrappers for the loaded public static methods. Publishing enables the Java methods to be called as PL/SQL functions or procedures.

Publishing Java to PL/SQL

```
FormatCreditCardNo.java   CCFORMAT

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

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

ORACLE

Publishing Java to PL/SQL

  The slide shows the Java code and how to publish the Java code in a PL/SQL procedure.

Creating Program Units

Skeleton of the function

Creating Program Units

To create a PL/SQL program unit:
1. Select View > Connection Navigator.
2. Expand Database and select a database connection.
3. In the connection, expand a schema.
4. Right-click a folder corresponding to the object type (Procedures, Packages, Functions).
5. Choose New PL/SQL object_type. The Create PL/SQL dialog box appears for the function, package, or procedure.
6. Enter a valid name for the function, package, or procedure and click OK.

A skeleton definition will be created and opened in the Code Editor. You can then edit the subprogram to suit your need.

Compiling
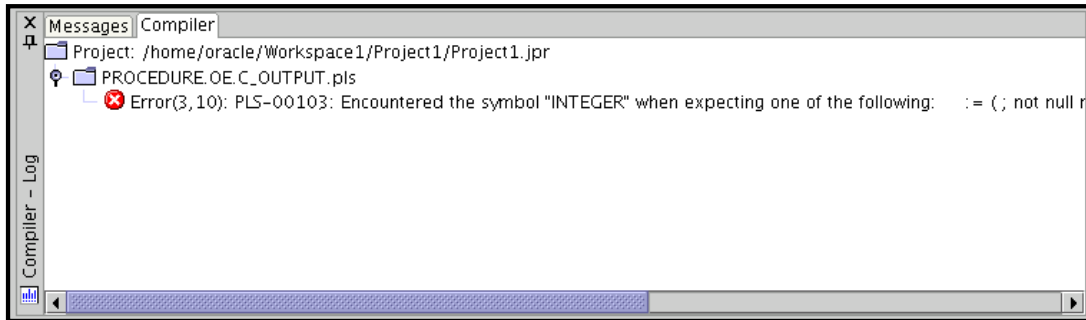
Compilation with errors
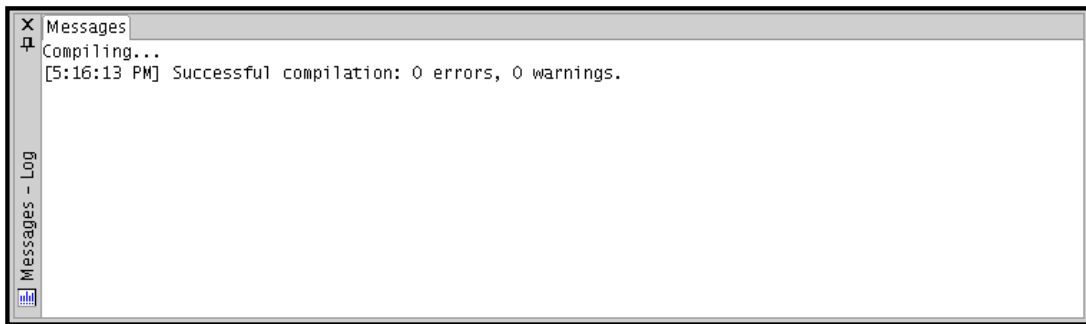
Compilation without errors

Compiling

After editing the skeleton definition, you need to compile the program unit. Right-click the PL/SQL object that you need to compile in the Connection Navigator and then select Compile. Alternatively you can also press CTRL + SHIFT + F9 to compile.

Running a Program Unit
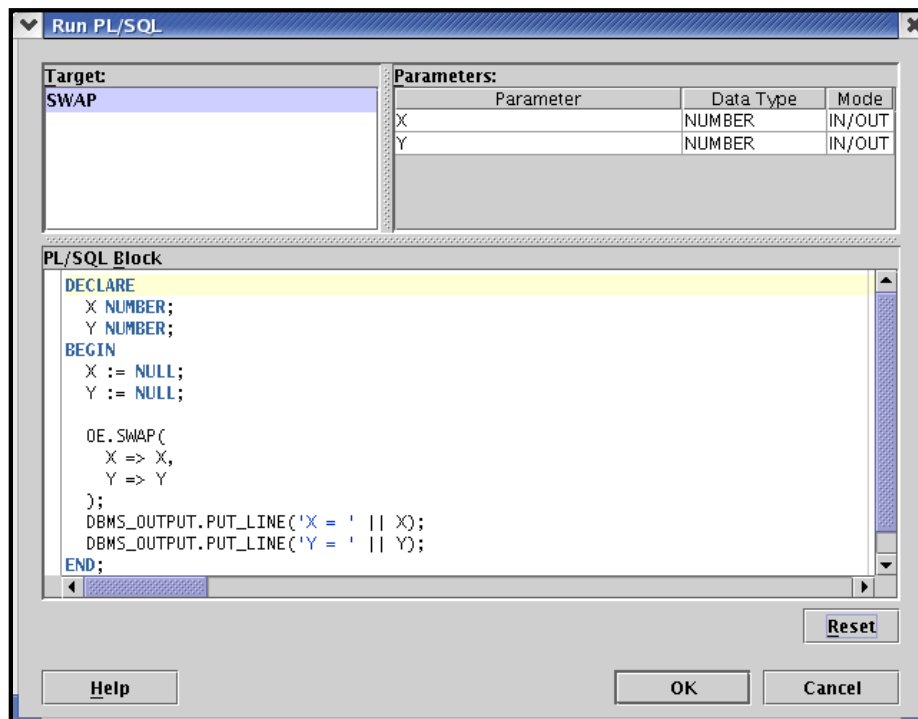
Running a Program Unit

To execute the program unit, right-click the object and click Run. The Run PL/SQL dialog box will appear. You may need to change the NULL values with reasonable values that are passed into the program unit. After you change the values, click OK. The output will be displayed in the Message-Log window.

Dropping a Program Unit
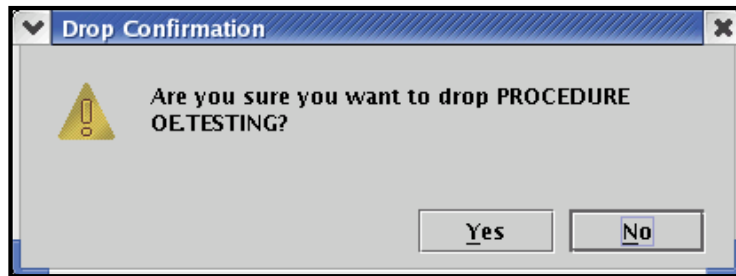
Dropping a Program Unit

To drop a program unit, right-click the object and select Drop. The Drop Confirmation dialog box will appear; click Yes. The object will be dropped from the database.

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

JDeveloper offers both local and remote debugging. A local debugging session is started by setting breakpoints in source files, and then starting the debugger. Remote debugging requires two JDeveloper processes: a `debugger` and a `debuggee` which may reside on a different platform.

To debug a PL/SQL program it must be compiled in `INTERPRETED` mode. You cannot debug a PL/SQL program that is compiled in `NATIVE` mode. This mode is set in the database's `init.ora` file.

PL/SQL programs must be compiled with the `DEBUG` option enabled. This option can be enabled using various ways. Using SQL*Plus, execute `ALTER SESSION SET PLSQL_DEBUG = true` to enable the `DEBUG` option. Then you can create or recompile the PL/SQL program you want to debug. Another way of enabling the `DEBUG` option is by using the following command in SQL*Plus:

```
ALTER <procedure, function, package> <name> COMPILE DEBUG;
```

Debugging PL/SQL Programs
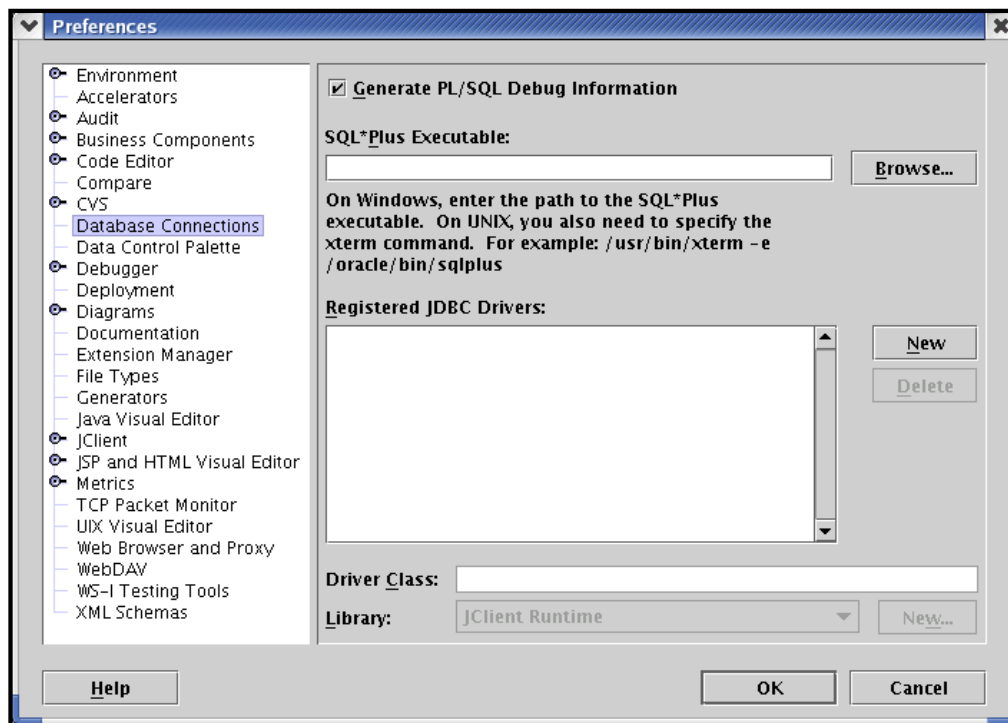
Debugging PL/SQL Programs (continued)

Before you start with debugging, make sure that the Generate PL/SQL Debug Information check box is selected. You can access the dialog box by using Tools > Preferences > Database Connections.

Instead of manually testing PL/SQL functions and procedures as you may be accustomed to doing from within SQL*Plus or by running a dummy procedure in the database, JDeveloper enables you to test these objects in an automatic way. With this release of JDeveloper, you can run and debug PL/SQL program units. For example, you can specify parameters being passed or return values from a function giving you more control over what is run and providing you output details about what was tested.

Note: The procedures or functions in the Oracle database can be either stand-alone or within a package.

Debugging PL/SQL Programs (continued)

To run or debug functions, procedures, and packages:
1. Create a database connection using the Database Wizard.
2. In the Navigator, expand the Database node to display the specific database username and schema name.
3. Expand the Schema node.
4. Expand the appropriate node depending on what you are debugging: Procedure, Function, or Package body.
5. (Optional for debugging only) Select the function, procedure, or package that you want to debug and double-click to open it in the Code Editor.
6. (Optional for debugging only) Set a breakpoint in your PL/SQL code by clicking to the left of the margin.

Note: The breakpoint must be set on an executable line of code. If the debugger does not stop, the breakpoint may have not been set on an executable line of code (verify that the breakpoint was verified). Also, verify that the debugging PL/SQL prerequisites were met. In particular, make sure that the PL/SQL program is compiled in the INTERPRETED mode.

7. Make sure that either the Code Editor or the procedure in the Navigator is currently selected.
8. Click the Debug toolbar button, or, if you want to run without debugging, click the Run toolbar button.
9. The Run PL/SQL dialog box is displayed.
   - Select a target that is the name of the procedure or function that you want to debug. Note that the content in the Parameters and PL/SQL Block boxes change dynamically when the target changes.

Note: You will have a choice of target only if you choose to run or debug a package that contains more than one program unit.

   • The Parameters box lists the target's arguments (if applicable).
   • The PL/SQL Block box displays code that was custom generated by JDeveloper for the selected target. Depending on what the function or procedure does, you may need to replace the NULL values with reasonable values so that these are passed into the procedure, function, or package. In some cases, you may need to write additional code to initialize values to be passed as arguments. In this case, you can edit the PL/SQL block text as necessary.

10. Click OK to execute or debug the target.
11. Analyze the output information displayed in the Log window.

In the case of functions, the return value will be displayed. DBMS_OUTPUT messages will also be displayed.
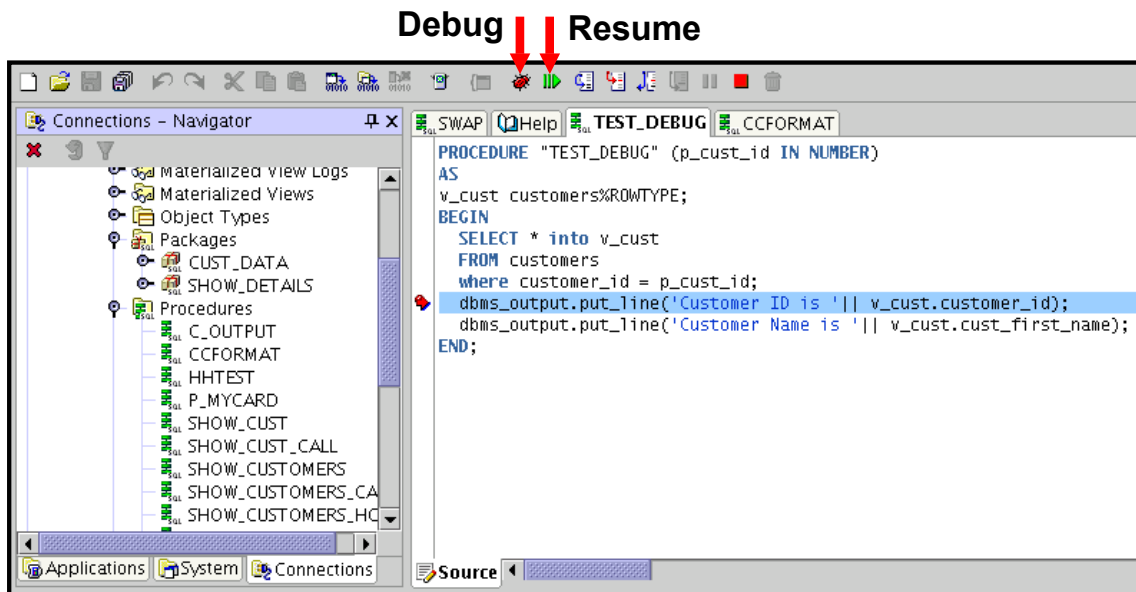
Setting Breakpoints

Setting Breakpoints

Breakpoints help you to examine the values of the variables in your program. It is a trigger in a program that, when reached, pauses program execution allowing you to examine the values of some or all of the program variables. By setting breakpoints in potential problem areas of your source code, you can run your program until its execution reaches a location you want to debug. When your program execution encounters a breakpoint, the program pauses, and the debugger displays the line containing the breakpoint in the Code Editor. You can then use the debugger to view the state of your program. Breakpoints are flexible in that they can be set before you begin a program run or at any time while you are debugging.

To set a breakpoint in the code editor, click the left margin next to a line of executable code. Breakpoints set on comment lines, blank lines, declaration and any other non-executable lines of code are not verified by the debugger and are treated as invalid.

Stepping Through Code

Debug **Resume**

Stepping Through Code

After setting the breakpoint, start the debugger by clicking the Debug icon. The debugger will pause the program execution at the point where the breakpoint is set. At this point you can check the values of the variables. You can continue with the program execution by clicking the Resume icon. The debugger will then move on to the next breakpoint. After executing all the breakpoints, the debugger will stop the execution of the program and display the results in the Debugging – Log area.

Examining and Modifying Variables
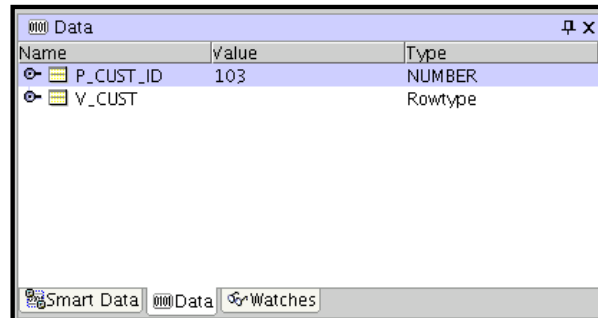
Data window

Examining and Modifying Variables

When the debugging is ON, you can examine and modify the value of the variables using the Data, Smart Data, and Watches windows. You can modify program data values during a debugging session as a way to test hypothetical bug fixes during a program run. If you find that a modification fixes a program error, you can exit the debugging session, fix your program code accordingly, and recompile the program to make the fix permanent.

You use the Data window to display information about variables in your program. The Data window displays the arguments, local variables, and static fields for the current context, which is controlled by the selection in the Stack window. If you move to a new context, the Data window is updated to show the data for the new context. If the current program was compiled without debug information, you will not be able to see the local variables.

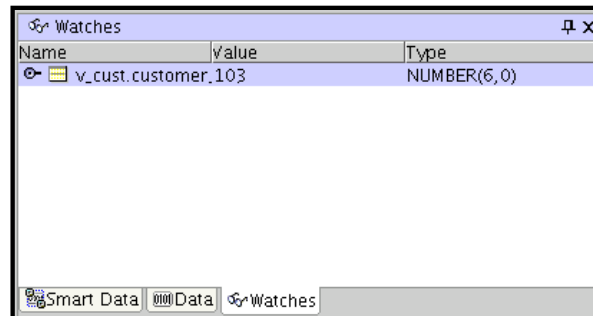# Examining and Modifying Variables
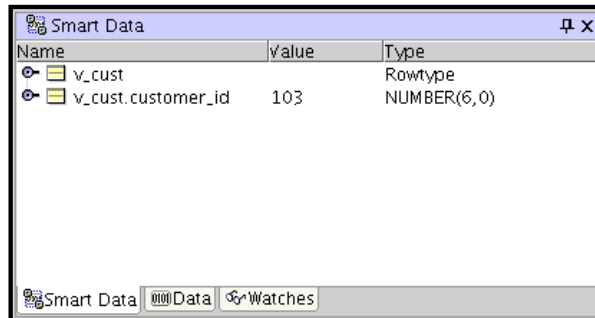
Smart Data window

Examining and Modifying Variables (continued)

Unlike the Data window that displays all the variables in your program, the Smart Data window displays only the data that is relevant to the source code that you are stepping through.
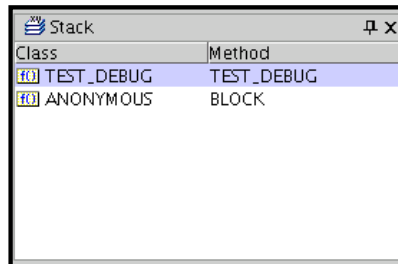
Examining and Modifying Variables

Watches window

Examining and Modifying Variables (continued)

A watch enables you to monitor the changing values of variables or expressions as your program runs. After you enter a watch expression, the Watch window displays the current value of the expression. As your program runs, the value of the watch changes as your program updates the values of the variables in the watch expression.

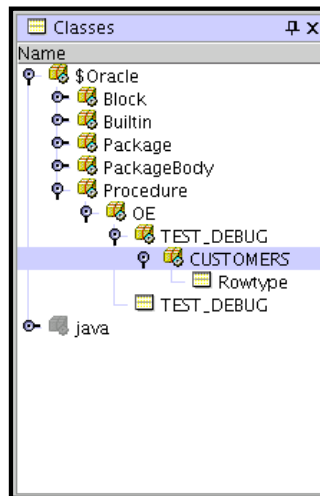# Examining and Modifying Variables

Stack window

Examining and Modifying Variables (continued)

You can activate the Stack window by using View > Debugger > Stack. It displays the call stack for the current thread. When you select a line in the Stack window, the Data window, Watch window, and all other windows are updated to show data for the selected class.

Examining and Modifying Variables

Classes window

Examining and Modifying Variables (continued)

The Classes window displays all the classes that are currently being loaded to execute the program. If used with Oracle Java Virtual Machine (OJVM), it also shows the number of instances of a class and the memory used by those instances.

D

Data Type Mappings

PL/SQL Data Type	Supported External Types	Default External Type
BINARY_INTEGER BOOLEAN PLS_INTEGER	[UNSIGNED] CHAR [UNSIGNED] SHORT [UNSIGNED] INT [UNSIGNED] LONG SB1, SB2, SB4 UB1, UB2, UB4 SIZE_T	INT
NATURALFoot 1 NATURALNFootref 1 POSITIVEFootref 1 POSITIVENFootref 1 SIGNTYPEFootref 1	[UNSIGNED] CHAR [UNSIGNED] SHORT [UNSIGNED] INT [UNSIGNED] LONG SB1, SB2, SB4 UB1, UB2, UB4 SIZE_T	UNSIGNED INT
FLOAT REAL	FLOAT	FLOAT
DOUBLE PRECISION	DOUBLE	DOUBLE
CHAR CHARACTER LONG NCHAR NVARCHAR2 ROWID VARCHAR VARCHAR2	STRING OCISTRING	STRING
LONG RAW RAW	RAW OCIRAW	RAW
BFILE BLOB CLOB NCLOB	OCILOBLOCATOR	OCILOBLOCATOR
NUMBER DECFootref 1	OCINUMBER	OCINUMBER

Oracle Database 10*g*: Advanced PL/SQL   D-2

PL/SQL Data Type	Supported External Types	Default External Type
DECIMAL INT INTEGER NUMERIC SMALLINT		
DATE	OCIDATE	OCIDATE
TIMESTAMP TIMESTAMP WITH TIME ZONE TIMESTAMP WITH LOCAL TIME ZONE	OCIDateTime	OCIDateTime
INTERVAL DAY TO SECOND INTERVAL YEAR TO MONTH	OCIInterval	OCIInterval
composite object types: collections (varrays, nested tables)	OCICOLL	OCICOLL

Oracle Database 10*g*: Advanced PL/SQL D-3

Index

Note: A bolded number or letter refers to an entire lesson or appendix.

Index

D

DAD   5-11, 5-13

Data type conversion   4-13, 4-22, 7-18

DBMS_DESCRIBE   8-3, 8-8, 8-9, 8-11, 8-12

DBMS_SHARED_POOL   7-25, 7-26, 7-36, 7-37

DBMS_TRACE   8-20, 8-21, 8-23, 8-25, 8-26, 8-29, 8-37

DBMS_UTILITY   8-3, 8-13, 8-15, 8-16

DESCRIBE_PROCEDURE   8-3, 8-8, 8-9

Directive   1-12, 1-27, 5-4, 5-7, 5-13, 5-18

E

Encapsulate   3-3, 7-5, 7-7, 7-22

Error stack   8-15, 8-19

Exception   1-2, 1-3, 1-6, 1-7, 1-11, 1-12, 1-13, 1-14, 1-21, 1-22, 1-23, 1-24, 1-27, 1-28, 1-29, 1-30, 1-38, 2-7, 2-17, 2-29, 3-25, 3-26, 3-39, 4-5, 4-6, 4-7, 4-8, 7-3, 7-14, 7-15, 7-20, 7-21, 8-3, 8-5, 8-15, 8-16, 8-19, 8-22, 8-27

External Routines   **4**

Extproc   4-5, 4-6, 4-7, 4-8

Index

F

Fine-grained access **6**

FORALL   7-2, 7-3, 7-9, 7-12, 7-14, 7-15

Function syntax   1-6

G

H

I

Implementing a policy   6-13, 6-14, 6-16, 6-18, 6-19

Initializing collections   3-18, 3-19

Interpreted compilation   7-2, 7-32, 7-34

J

Java   **4**

Jdeveloper   **C**

K

L

Library   4-3, 4-5, 4-6, 4-7, 4-8, 4-9, 4-10, 4-12, 4-14, 4-16, 4-17, 4-19,
                7-19, 7-32, 8-30

Libunit   4-3, 4-19, 4-21

Listener   4-5, 4-6, 4-7, 4-8

Logon trigger   6-13, 6-19, 6-27

Oracle Database 10*g*: Advanced PL/SQL Index-3

Index

M

Memory issues   7-2, 7-24

Mutually exclusive conditions   7-17

N

Native compilation   7-32, 7-34, 7-36

Nested table storage   3-15

Nested tables   3-8, 3-9, 3-11, 3-12, 3-15, 3-20, 3-31, 3-32, 3-33, 3-39, 7-8

O

Object methods   3-4

Object type columns   3-7

Object types   3-2, 3-3, 3-4, 3-5, 3-8, 3-13

P

Package syntax   1-11, 1-13, 1-14

Passing records   7-22

Persistent objects   3-3

Pinning   7-25, 7-26, 7-28, 7-37

`PLSQL_TRACE_EVENTS`   8-28, 8-29

`PLSQL_TRACE_RUNS`   8-28, 8-29

PL/SQL basic syntax   **1**

PL/SQL server pages   **5**

Policy   6-3, 6-4, 6-5, 6-6, 6-13, 6-14, 6-16, 6-18, 6-19, 6-22, 6-23, 6-24, 6-27

Policy groups   6-4, 6-23

`POST`   3-3, 3-4, 3-7, 5-11, 5-16, 7-20

`PRAGMA`   1-12, 1-27, 4-13, 7-15

Procedure syntax   1-5

Profiling   8-2, 8-24, 8-30, 8-31, 8-32, 8-33, 8-34, 8-35, 8-36, 8-38

Index

Q

Querying collections   3-27, 3-28

R

Reducing network traffic   7-29, 7-30

REF CURSOR   2-8, 2-10, 2-11, 2-12, 2-13, 2-14, 2-18, 2-19, 2-21

Referencing collection elements   3-20

RETURNING clause   7-5, 7-12, 7-30, 7-36

S

Scriptlet   5-4, 5-7

Set operations   3-31, 3-32, 3-33, 3-34

Setting a context   6-11

SGA   7-7, 7-24

Shared pool   7-2, 7-24, 7-25, 7-26, 7-28, 7-36

Smaller executable sections   7-2, 7-3, 7-4

SQLERRM   7-15, 8-15, 8-16

Start tracing   8-20, 8-23, 8-25

String indexed arrays   3-35, 3-36, 3-37

Subtypes   1-13, 1-14, 2-2, 2-23, 2-24, 2-26, 2-27, 2-28, 2-29, 2-30, 2-31, 7-19, 7-20, 8-12

SYS_CONTEXT   6-6, 6-9

Index

T

Table expression   3-27, 3-28, 3-30

Trace information   8-19, 8-23, 8-24, 8-26, 8-27, 8-28, 8-29

Trace level   8-22, 8-23, 8-25, 8-27

Transient objects   3-3

Traversing collections   3-22

Tuning issues   **7**

U

V

Varrays   3-8, 3-9, 3-11, 3-12, 3-20, 3-31, 3-39, 7-8

Virtual Private Database (VPD)   **6**

W

X

Y

Z