Oracle Database 12*c*: Advanced PL/SQL

Student Guide - Volume I

D80343GC10
Edition 1.0
April 2014
D86295

ORACLE

Author

Sharon Sophia Stephen

Technical Contributors and Reviewers

Branislav Valny

Brent Dayley

Krishnanjani Chitta

Laszlo Czinkoczki

Nancy Greenberg

Sailaja Pasupuleti

Swarnapriya Shridhar

Wayne Abbott

Editors

Aju Kumar

Raj Kumar

Graphic Designer

Divya Thallap

Publishers

Joseph Fernandez

Jayanthy Keshavamurthy

Veena Narasimhan

Contents

1 Introduction

ORACLE

Course Objectives

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

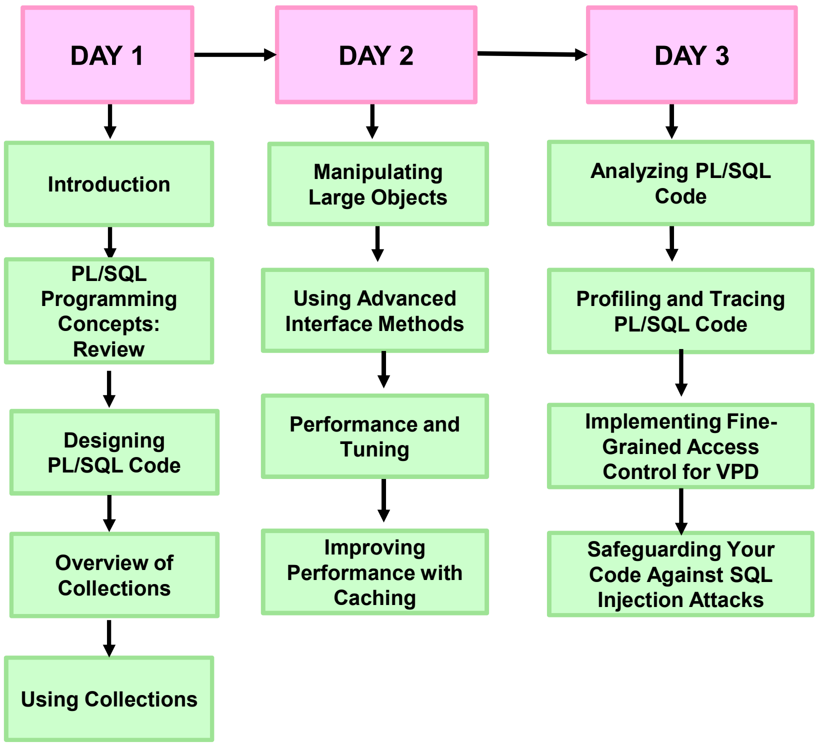
- Design PL/SQL packages and program units that execute efficiently
- Write code to interface with external applications and the operating system
- Create PL/SQL applications that use collections
- Write and tune PL/SQL code effectively to maximize performance
- Implement a virtual private database with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

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 the 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, and fine-grained access are covered in this course.

Lesson Agenda

- Previewing the course agenda
- Describing the development environments
- Identifying the tables and schemas used in this course

Course Agenda


DAY 1
DAY 2
DAY 3
Introduction
PL/SQL Programming Concepts: Review
Designing PL/SQL Code
Overview of Collections
Using Collections
Manipulating Large Objects
Using Advanced Interface Methods
Performance and Tuning
Improving Performance with Caching
Analyzing PL/SQL Code
Profiling and Tracing PL/SQL Code
Implementing Fine-Grained Access Control for VPD
Safeguarding Your Code Against SQL Injection Attacks

In this three-day course, you start with a review of PL/SQL concepts before progressing into the new and advanced topics. By the end of each day, you should cover the following:

- Day one
  - Design considerations for your program units
  - How to use collections effectively
- Day two
  - How to use advanced interface methods to call C and Java code from your PL/SQL programs
  - How to manipulate large objects programmatically through PL/SQL
  - How to administer the features of the new LOB formats
  - How to tune PL/SQL code and deal with memory issues
- Day three
  - How to improve performance by using Oracle database 12*c* caching techniques
  - How to write PL/SQL routines that analyze PL/SQL applications
  - How to profile and trace PL/SQL code
  - How to implement and test fine-grained access control for virtual private databases
  - How to protect your code from SQL injection security attacks

Appendixes Used in This Course

- Appendix A: Table Descriptions and Data
- Appendix B: Using SQL Developer
- Appendix C: Using SQL*Plus
- Appendix D: Designing and Testing Your Code to Avoid SQL Injection Attacks

Lesson Agenda

- Previewing the course agenda
- Describing the development environments
- Identifying the tables and schemas used in this course

Development Environments: Overview

- SQL Developer
- SQL*Plus

SQL Developer

This course has been developed using Oracle SQL Developer as the tool for running the SQL statements discussed in the examples in the slide and the practices.

- SQL Developer is shipped with Oracle Database, and is the default tool for this course.

SQL*Plus

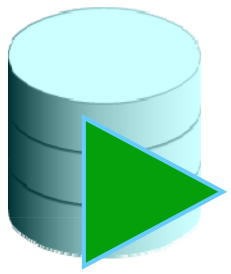
The SQL*Plus environment can also be used to run the SQL commands covered in this course.

Note

- See Appendix B titled “Using SQL Developer” for information about using SQL Developer, including simple instructions on installing.
- See Appendix C titled “Using SQL*Plus” for information about using SQL*Plus.

What Is Oracle SQL Developer?

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



SQL Developer

Oracle SQL Developer is a free graphical tool designed to improve your productivity and simplify the development of everyday database tasks. With just a few clicks, you can easily create and maintain stored procedures, test SQL statements, and view optimizer plans.

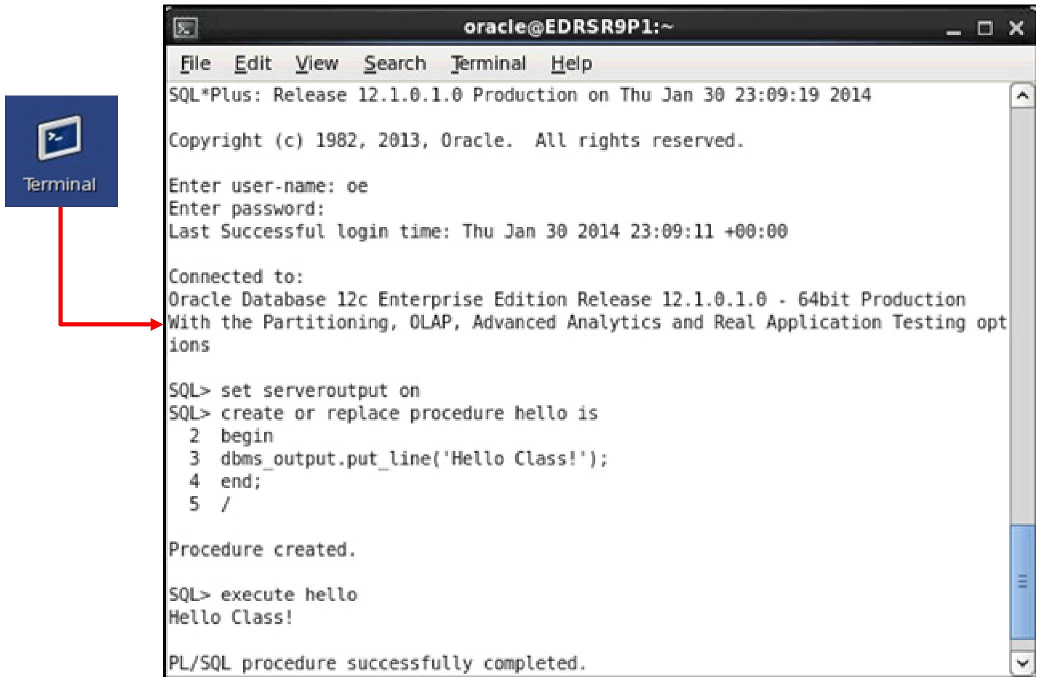
SQL Developer simplifies the following tasks:

- Browsing and managing database objects
- Executing SQL statements and scripts
- Editing and debugging PL/SQL statements
- Creating reports

You can connect to any target Oracle database schema using standard Oracle database authentication. When connected, you can perform operations on objects in the database.

For an introduction to using the SQL Developer interface, see Appendix B of this course.

Coding PL/SQL in SQL*Plus



```
SQL*Plus: Release 12.1.0.1.0 Production on Thu Jan 30 23:09:19 2014

Copyright (c) 1982, 2013, Oracle.  All rights reserved.

Enter user-name: oe
Enter password:
Last Successful login time: Thu Jan 30 2014 23:09:11 +00:00

Connected to:
Oracle Database 12c Enterprise Edition Release 12.1.0.1.0 - 64bit Production
With the Partitioning, OLAP, Advanced Analytics and Real Application Testing opt
ions

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

Procedure created.

SQL> execute hello
Hello Class!

PL/SQL procedure successfully completed.
```

Oracle SQL*Plus is a command-line interface that enables you to submit SQL statements and PL/SQL blocks for execution and receive the results in an application or a command window.

SQL*Plus is:

- Shipped with the database
- Accessed from an icon or the command line

When coding PL/SQL subprograms using SQL*Plus, remember the following:

- You create subprograms by using the `CREATE SQL` statement.
- You execute subprograms by using either an anonymous PL/SQL block or the `EXECUTE` command.
- If you use the `DBMS_OUTPUT` package procedures to print text to the screen, you must first execute the `SET SERVEROUTPUT ON` command in your session.

Note

- To launch SQL*Plus in the Linux environment, open a Terminal window and enter the `sqlplus` command.
- For more information about how to use SQL*Plus, refer to Appendix C in this course.

Lesson Agenda

- Previewing the course agenda
- Describing the development environments
- Identifying the tables and schemas used in this course

Tables Used in This Course

- The sample schemas used are:
  - Order Entry (`OE`) schema
  - Human Resources (`HR`) schema
- Primarily, the `OE` schema is used.
- The `OE` schema user can read data in the `HR` schema tables.
- Appendix A contains more information about the sample schemas.

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

- The Human Resources division tracks information about the employees and the 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. Although not used in this course, the `SH` schema is part of the "Example" sample schemas shipped with the database.

Each of these divisions is represented by a schema.

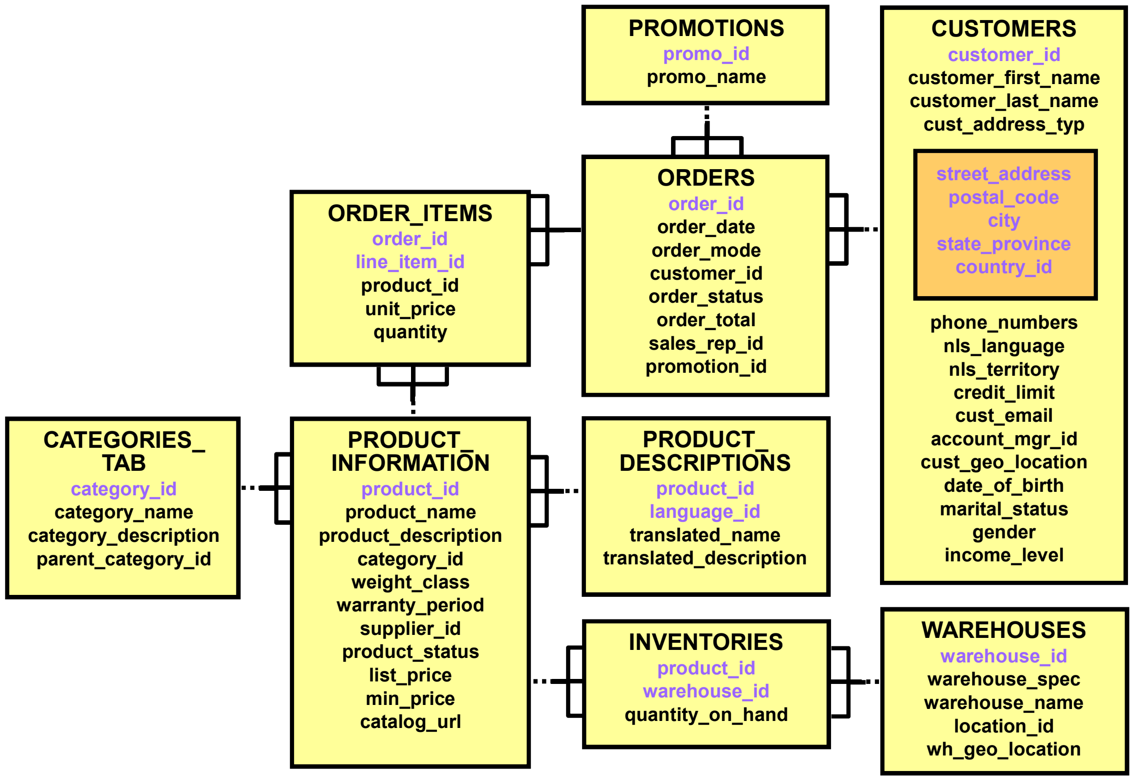
This course primarily uses the Order Entry (`OE`) sample schema.

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

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.

Order Entry 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 different languages.

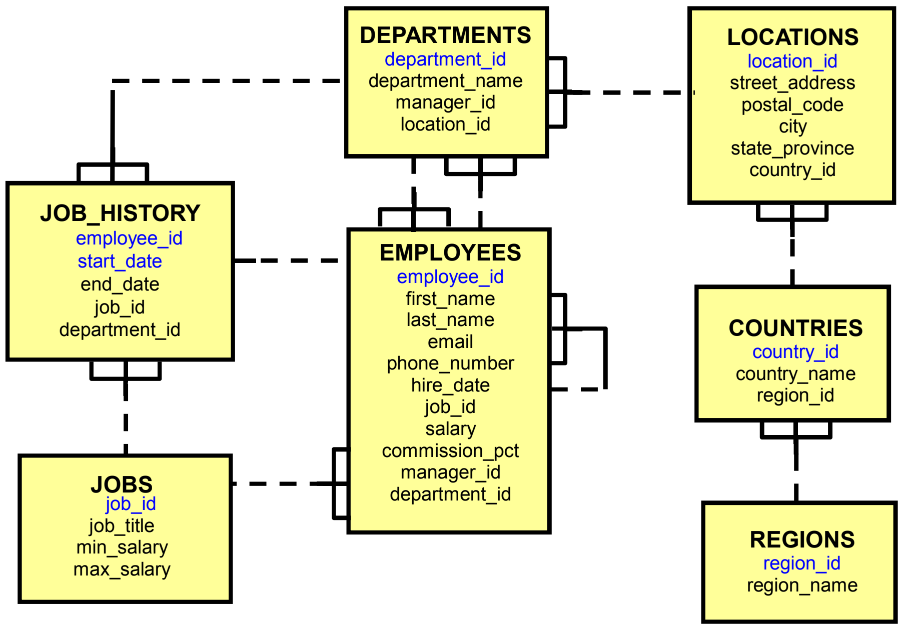
The company maintains warehouses in several locations to facilitate filling customer orders. Each warehouse has a warehouse identification number, name, and location identification number.

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 email 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. It keeps track of a customer's phone number. At present, you do not know how many phone numbers a customer might have, but you try to keep track of all of them. Because of the language differences among customers, you also 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 who are living in different geographic regions of the world.

Human Resources Schema



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

The company also tracks information about the 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.

Note: For more information about the “Example” sample schemas, refer to Appendix A.

Oracle SQL and PL/SQL Documentation

- *Oracle Database New Features Guide*
- *Oracle Database Advanced Application Developer's Guide*
- *Oracle Database PL/SQL Language Reference*
- *Oracle Database Reference*
- *Oracle Database SQL Language Reference*
- *Oracle Database Concepts*
- *Oracle Database PL/SQL Packages and Types Reference*
- *Oracle Database SQL Developer User's Guide*

Summary

In this lesson, you should have learned how to:

- Describe the goals of the course
- Identify the environments that can be used in this course
- Describe the database schema and tables that are used in the course
- List the available documentation and resources

In this lesson, you were introduced to the goals of the course, the SQL Developer and SQL*Plus environments used in the course, and the database schema and tables used in the lectures and practices.

Practice 1 Overview: Getting Started

This practice covers the following topics:

- Reviewing the available SQL Developer resources
- Starting SQL Developer, and creating new database connections and browsing the `HR`, `OE`, and `SH` tables
- Executing SQL statements and an anonymous PL/SQL block by using SQL Worksheet
- Accessing and bookmarking the Oracle Database documentation and other useful websites

In this practice, you use SQL Developer to execute SQL statements for examining the data in the "Example" sample schemas: `HR`, `OE`, and `SH`. You also create a simple anonymous block. Optionally, you can experiment by creating and executing the PL/SQL code in SQL*Plus.

Note: All written practices use SQL Developer as the development environment. Although it is recommended that you use SQL Developer, you can also use the SQL*Plus environment that is available in this course.

2

PL/SQL Programming Concepts: Review

Objectives

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

- Describe PL/SQL basics
- List restrictions on calling functions from SQL expressions
- Review PL/SQL packages
- Identify how explicit cursors are processed
- Handle exceptions
- Use the `RAISE_APPLICATION_ERROR` procedure
- Manage dependencies
- Use Oracle-supplied packages

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.

Pre-Quiz

What are the named PL/SQL blocks?

Answer: Procedures and Functions

Pre-Quiz

A PL/SQL block must consist of the following three sections:

- A Declarative section, which begins with the keyword DECLARE and ends when the executable section starts
- An executable section, which begins with the keyword BEGIN and ends with END
- An Exception handling section, which begins with the keyword EXCEPTION and is nested within the executable section

a. True

b. False

Answer: b

A PL/SQL block consists of three sections:

- Declarative (optional): The optional declarative section begins with the keyword DECLARE and ends when the executable section starts.
- Executable (required): The required executable section begins with the keyword BEGIN and ends with END. This section essentially needs to have at least one statement. Observe that END is terminated with a semicolon. The executable section of
- A PL/SQL block can, in turn, include any number of PL/SQL blocks.
- Exception handling (optional): The optional exception section is nested within the executable section. This section begins with the keyword EXCEPTION.

Pre-Quiz

An exception handling section can contain nested blocks.

a. True

b. False

Answer: a

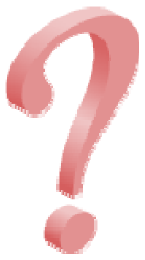
You can nest blocks wherever an executable statement is allowed, thus making the nested block a statement. The exception section can also contain nested blocks.

Note: An exception section can also contain another exception.

Pre-Quiz

All PL/SQL statements are processed in:

a. Procedural Statement Executor
b. SQL Statement Executor
c. m-code Processor
d. None of the above



Answer: a

All PL/SQL statements are processed in the Procedural Statement Executor, and all SQL statements must be sent to the SQL Statement Executor for processing by the Oracle Server processes.

Pre-Quiz

If you use the `DBMS_OUTPUT` package procedures to print text to the screen, you must first execute the `SET SERVEROUTPUT ON` command in your session.

a. True

b. False

Answer: a

To enable output to be displayed, execute the `SET SERVEROUTPUT ON` command before running the PL/SQL block.

It will enable the buffer with its default size of 2000 bytes for displaying the output.

Use the predefined Oracle package `DBMS_OUTPUT` and its procedure `PUT_LINE` in the anonymous block.

Pre-Quiz

What are the special characters allowed in variable names?

Answer: $, _, and #

Pre-Quiz

Why are bind variables called host variables?

a. You create bind variables in a host environment.
b. You put your data in a host variable.
c. Both a and b
d. None of the above

Answer: c

Pre-Quiz

Identify the functions that are not available in procedural statements.

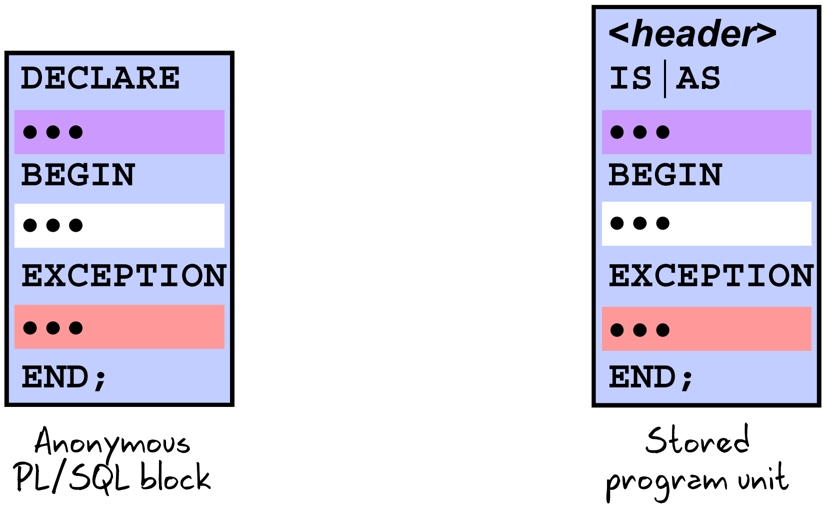
a. `DECODE`
b. `VARIANCE`
c. `COUNT`
d. `LENGTH`
e. `TO_DATE`

Answer: a, b, and c

Lesson Agenda

- Describing PL/SQL basics
- Listing restrictions on calling functions from SQL expressions
- Reviewing PL/SQL packages
- Identifying how explicit cursors are processed
- Handling exceptions
- Using the `RAISE_APPLICATION_ERROR` procedure
- Managing dependencies
- Using Oracle-supplied packages

PL/SQL Block Structure



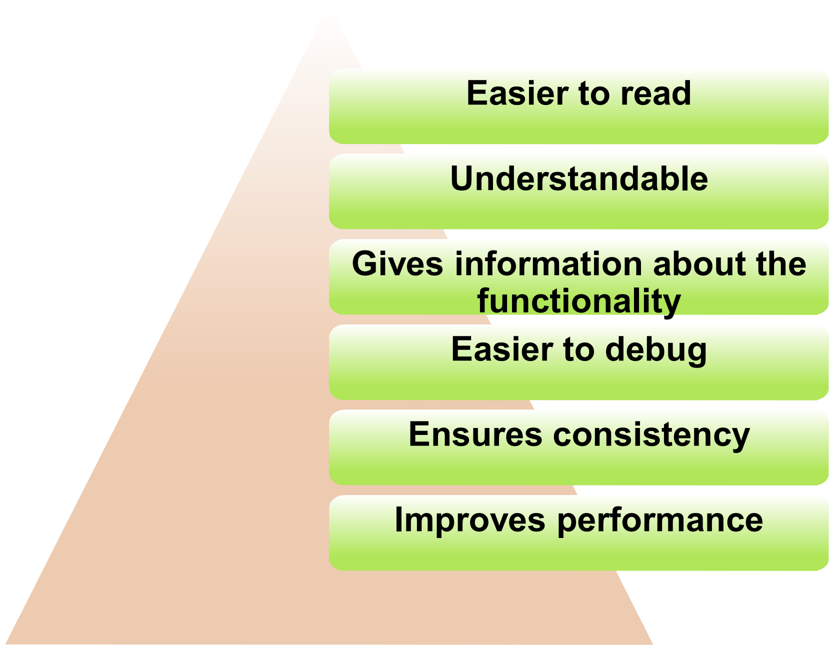
An anonymous PL/SQL block structure consists of an optional `DECLARE` section, a mandatory `BEGIN-END` block, and an optional `EXCEPTION` section before the `END` statement of the main block.

A stored program unit:

- Has a mandatory header section, which defines whether the program unit is a function, procedure, or package, and contains the optional argument list and its modes. It also has the other sections mentioned for the anonymous PL/SQL block.
- Does not have an optional `DECLARE` section. It contains a mandatory `IS | AS` section that acts like the `DECLARE` section in an anonymous 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:



A proper naming convention makes the code easier to read and more understandable. It helps you to 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.

Naming Conventions

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`

The table in this slide shows the naming conventions followed in this course.

Procedures

A procedure is:

- A named PL/SQL block that performs a sequence of actions and optionally returns a value or values
- 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];
```

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 declarative 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 by using the same parameter.

Procedure: Example

```
CREATE OR REPLACE PROCEDURE get_avg_order
(p_cust_id NUMBER,
 p_cust_last_name VARCHAR2,
 p_order_tot NUMBER)
IS
   v_cust_ID customers.customer_id%type;
   v_cust_name customers.cust_last_name%type;
   v_avg_order NUMBER;
BEGIN
SELECT customers.customer_id, customers.cust_last_name,
AVG(orders.order_total)
INTO v_cust_id, v_cust_name, v_avg_order
FROM CUSTOMERS, ORDERS
WHERE customers.customer_id=orders.customer_id
GROUP BY customers.customer_id, customers.cust_last_name;
END;
/
```

This reusable procedure has a parameter with a `SELECT` statement for getting average order totals for whatever customer value is passed in.

Note: If a developer drops a procedure, and then re-creates it, all applicable grants to execute the procedure are lost. Alternatively, the `OR REPLACE` command removes the old procedure and re-creates it but leaves all the grants against the said procedure in place. Thus, the `OR REPLACE` command is recommended wherever there is an existing procedure, function, or package; not merely for convenience, but also to protect granted privileges. If you grant object privileges, these privileges remain after you re-create the subprogram with the `OR REPLACE` option; otherwise, the privileges are not preserved.

Stored Functions

A function is:

- A named block that must return 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];
```

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, and must return a value in each exception handler to avoid the "ORA-06503: PL/SQL: Function returned without value" error.

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 must be updated.

A function can 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.

Functions: 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))
```

The `get_credit` function is created with a single input parameter and returns the credit limit as a number, as shown in the first code box in the slide. 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, it may also contain a `RETURN` statement.

Invoke a function as part of a PL/SQL expression, because the function returns a value to the calling environment. The second code box in the slide 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, `DBMS_OUTPUT.PUT_LINE` is invoked first; it calls `get_credit` 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 then displays the result (if you have executed `SET SERVEROUTPUT ON`).

Note: The `%TYPE` attribute casts the data type to the type defined for the column in the table identified. You can use the `%TYPE` attribute as a data type specifier when declaring constants, variables, fields, and parameters.

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;
```

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 12c: SQL Fundamentals I* course.
- **As an expression in a SQL statement:** The last example in the slide shows how a function can be used as a single-row function in a SQL statement.

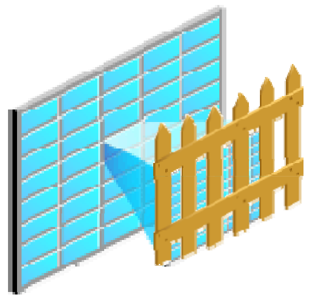
Note: The restrictions and guidelines that apply to functions when used in a SQL statement are discussed in the next few slides.

Lesson Agenda

- Describing PL/SQL basics
- Listing restrictions on calling functions from SQL expressions
- Reviewing PL/SQL packages
- Identifying how explicit cursors are processed
- Handling exceptions
- Using the `RAISE_APPLICATION_ERROR` procedure
- Managing dependencies
- Using Oracle-supplied packages

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:
  - You must own the function or have the `EXECUTE` privilege



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`. However, the return type can be `PLS_INTEGER` and `BINARY_INTEGER`. The same restriction applies to the parameters of the function.

The following restrictions apply when calling a function in a SQL statement:

- 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.

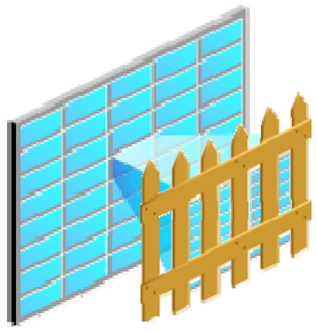
Note: Only stored functions are callable from SQL statements. Stored procedures cannot be called unless invoked from a function that meets the preceding requirements.

Restrictions on Calling Functions from SQL Expressions

Functions called from SQL statements cannot:

- Contain DML statements
- 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.



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

Additional restrictions also 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 a database table, unless the modification occurs in an autonomous transaction
- An `INSERT... SELECT` (but not an `INSERT... VALUES`), an `UPDATE`, or a `DELETE` statement, the function cannot query or modify a database table that was modified by that statement
- A `SELECT`, `INSERT`, `UPDATE`, or `DELETE` statement, the function cannot execute directly or indirectly through another subprogram or through 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 data definition language (DDL) statements (such as `CREATE`), because they are followed by an automatic commit

Note: The function *can* execute a transaction control statement if the transaction being controlled is autonomous.

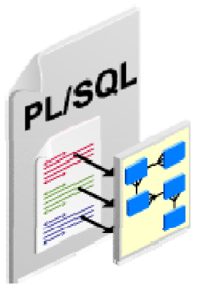
Lesson Agenda

- Describing PL/SQL basics
- Listing restrictions on calling functions from SQL expressions
- Reviewing PL/SQL packages
- Identifying how explicit cursors are processed
- Handling exceptions
- Using the `RAISE_APPLICATION_ERROR` procedure
- Managing dependencies
- Using Oracle-supplied packages

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 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 that are stored separately in the database:

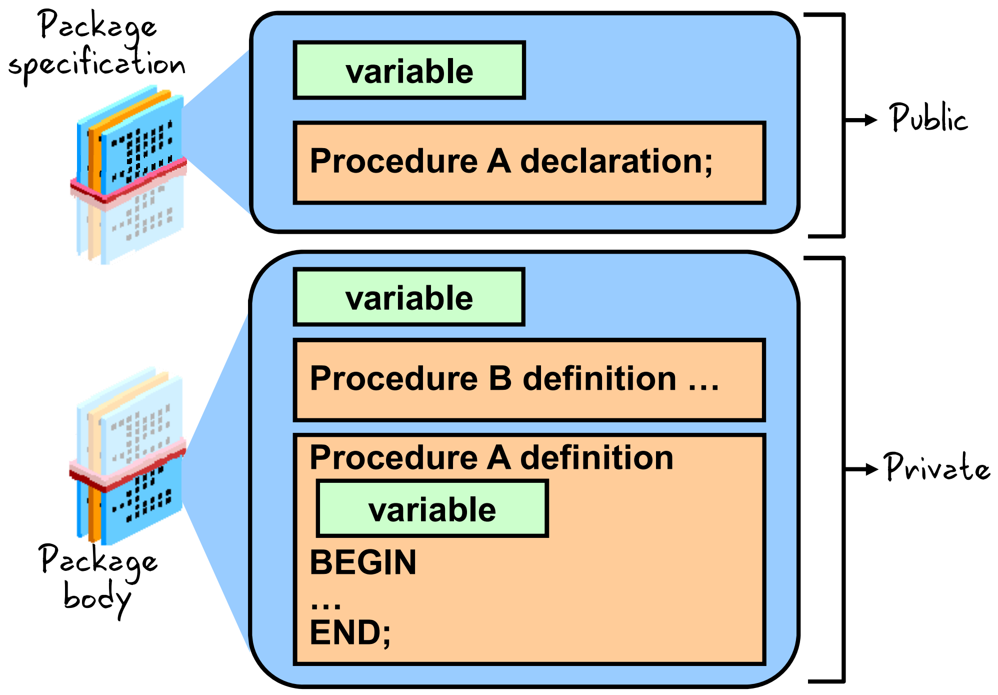
- A specification
- A body (optional)

Note: Having a package body is mandatory if the specification contains subprograms only.

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. However, subsequent access to constructs in the same package does not require disk I/O.

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 that are 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 the subprograms that are declared in the specification part. The package body can also define PL/SQL constructs, such as object types, variables, constants, exceptions, and cursors.

Public components are declared in the package specification. The specification defines a public API for users of the 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 but not referenced in the specification and can be referenced only by other constructs within the same package body. Alternatively, private components can reference the public components of the package.

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

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 constructs declared in a package specification are visible to users who are granted privileges on the package.

- 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 the 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 signatures terminated by a semicolon. The signature is everything above `IS|AS` keywords. 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.

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

- Specify the `OR REPLACE` option to overwrite a 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 the definitions of items in the package body syntax:

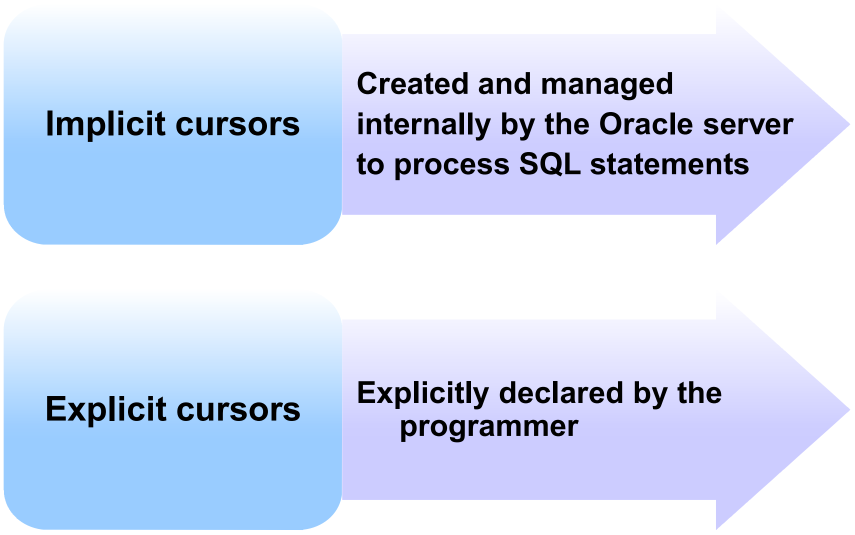
- ***`package_name`*** specifies a name for the package that must be the same as its package specification. Using the package name after the `END` keyword is optional.
- ***`private type and variable declarations`*** declares private variables, constants, cursors, exceptions, user-defined types, and 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.

Lesson Agenda

- Describing PL/SQL basics
- Listing restrictions on calling functions from SQL expressions
- Reviewing PL/SQL packages
- Identifying how explicit cursors are processed
- Handling exceptions
- Using the `RAISE_APPLICATION_ERROR` procedure
- Managing dependencies
- Using Oracle-supplied packages

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

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 by using the `INTO` clause.

Where Does Oracle Process SQL Statements?

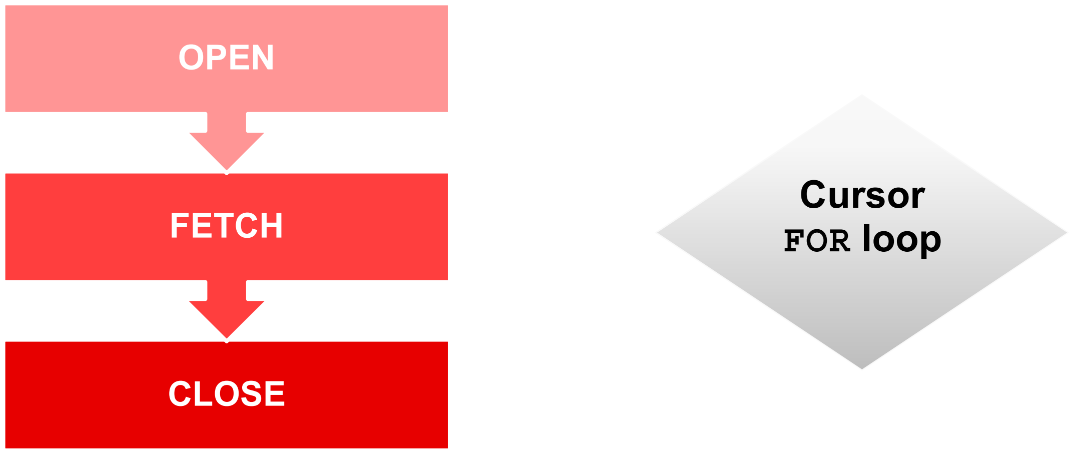
The Oracle server allocates a private memory area, called the *context area,* to process SQL statements. The SQL statement is parsed and processed in this area. The information required for processing and the information retrieved after processing are stored in this area. Because this area is internally 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. Oracle not only creates cursors for DML statements, but also for DDL and DCL statements, such as `CREATE`, `ALTER`, `DROP TABLE`, `GRANT REVOKE`, and so on. 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 executes a SQL statement, such as `SELECT`, `INSERT`, `UPDATE`, or `DELETE`.

- **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



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

- `OPEN`
- `FETCH`
- `CLOSE`

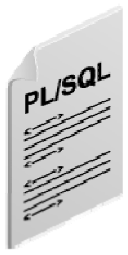
You initialize the cursor with the `OPEN` command, which recognizes the result set. Then, you execute the `FETCH` command repeatedly in a loop until all rows are retrieved. Alternatively, you can use a `BULK COLLECT` clause to fetch all rows at once. After the last row is processed, you release the cursor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

Note: You can also use a cursor `FOR` loop to process the query.

Explicit Cursor Attributes

Every explicit cursor has the following attributes:

- *cursor_name*%FOUND
- *cursor_name*%ISOPEN
- *cursor_name*%NOTFOUND
- *cursor_name*%ROWCOUNT



When cursor attributes are appended to the cursors, they return useful information about the execution of the data manipulation language (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.

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 in 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

```
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;
/
```

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

`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: An `INTO` clause or a `FETCH` statement is not required because the `FETCH INTO` is implicit. The code does not have `OPEN` and `CLOSE` statements to open and close the cursor, respectively.

Lesson Agenda

- Describing PL/SQL basics
- Listing restrictions on calling functions from SQL expressions
- Reviewing PL/SQL packages
- Identifying how explicit cursors are processed
- Handling exceptions
- Using the `RAISE_APPLICATION_ERROR` procedure
- Managing dependencies
- Using Oracle-supplied packages

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
  - By trapping and propagating it

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.

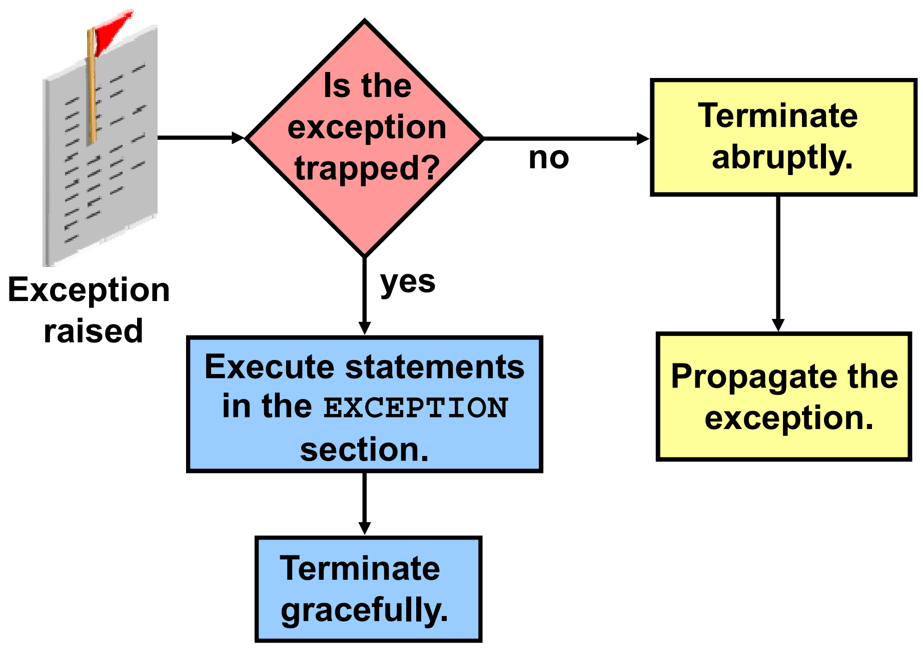
Methods for Raising an Exception

- 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, PL/SQL raises the `NO_DATA_FOUND` exception. These errors are converted into predefined exceptions.
- Depending on the business functionality that your program is implementing, you may have to explicitly raise an exception 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.

Methods for Handling an Exception

The third method in the slide for handling an exception involves trapping and propagating. It is often very important to be able to handle an exception after propagating it to the invoking environment, by issuing a simple `RAISE` statement.

Handling Exceptions



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, 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;
/
```

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 following is the output of this code:

```
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 statements in the specified `EXCEPTION` section are executed. The PL/SQL block terminates with normal, successful completion. Only one exception handler is executed.

Note the `SELECT` statement in the executable block. That statement requires that a query must return *only* one row. If multiple rows are returned, a “too many rows” exception is raised. If no rows are returned, a “no data found” exception is raised. The block of code in the slide tests for the “too many rows” exception.

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`

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 the *PL/SQL User's Guide and Reference*.

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

Predefined Oracle Server Exceptions

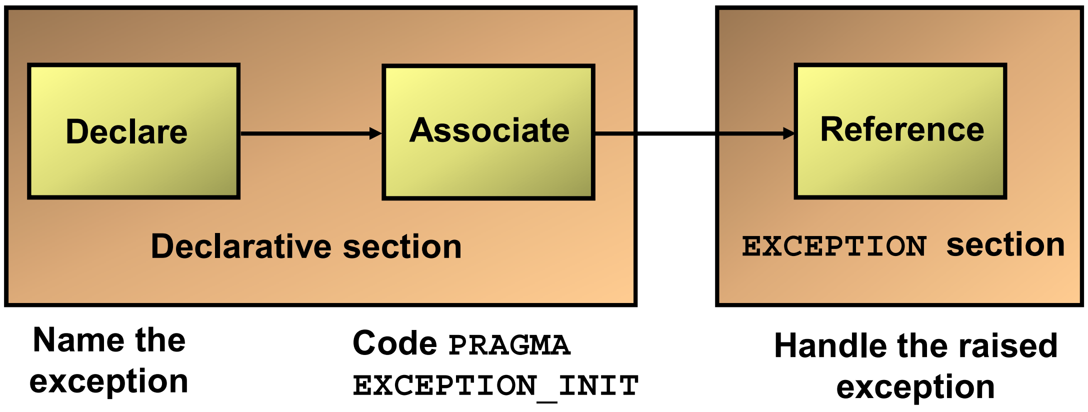
Exception Name	Oracle Server Error Number	Description
`ACCESS_INTO_NULL`	`ORA-06530`	Attempted to assign values to the attributes of an uninitialized object
`CASE_NOT_FOUND`	`ORA-06592`	None of the choices in the `WHEN` clauses of a `CASE` statement is selected, and there is no `ELSE` clause.
`COLLECTION_IS_NULL`	`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`	An Illegal cursor operation occurred.
`INVALID_NUMBER`	`ORA-01722`	Conversion of character string to number failed.
`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`	The PL/SQL program issued a database call without being connected to the Oracle server.

This slide and the following slide list some of the common exceptions.

Predefined Oracle Server Exceptions

Exception Name	Oracle Server Error Number	Description
`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.
`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 failed because the character string did 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



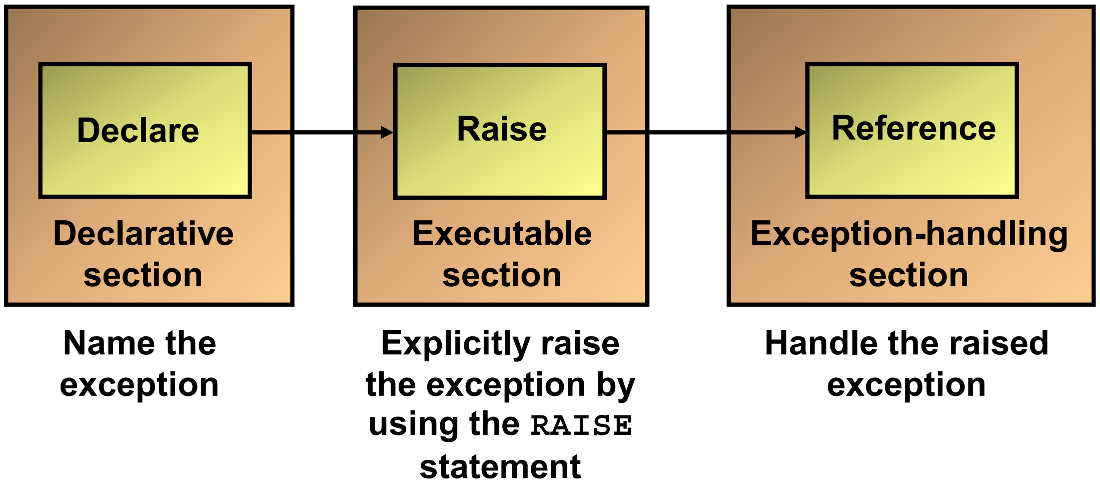
Non-predefined exceptions are similar to predefined exceptions; however, they are not defined as PL/SQL exceptions in the Oracle server. They are standard Oracle errors. You can create exceptions with standard Oracle errors by using the `PRAGMA EXCEPTION_INIT` function. Such exceptions are called non-predefined 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. This allows you to refer to any internal exception by name and to write a specific handler for it.

```
DECLARE
     e_MissingNull EXCEPTION;
    PRAGMA EXCEPTION_INIT(e_MissingNull, -1400);
    BEGIN
    INSERT INTO employees (id) VALUES (NULL);
    EXCEPTION
     WHEN e_MissingNull then
        DBMS_OUTPUT.put_line('ORA-1400 occurred');
    END;
 /
ORA-1400 occurred
```

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
Declarative section
Raise
Executable section
Reference
Exception-handling section
Name the exception
Explicitly raise the exception by using the RAISE statement
Handle the raised exception

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, 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

Lesson Agenda

- Describing PL/SQL basics
- Listing restrictions on calling functions from SQL expressions
- Reviewing PL/SQL packages
- Identifying how explicit cursors are processed
- Handling exceptions
- Using the `RAISE_APPLICATION_ERROR` procedure
- Managing dependencies
- Using Oracle-supplied packages

`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.

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 (This is not an Oracle-defined exception number.)
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.)

`RAISE_APPLICATION_ERROR` Procedure

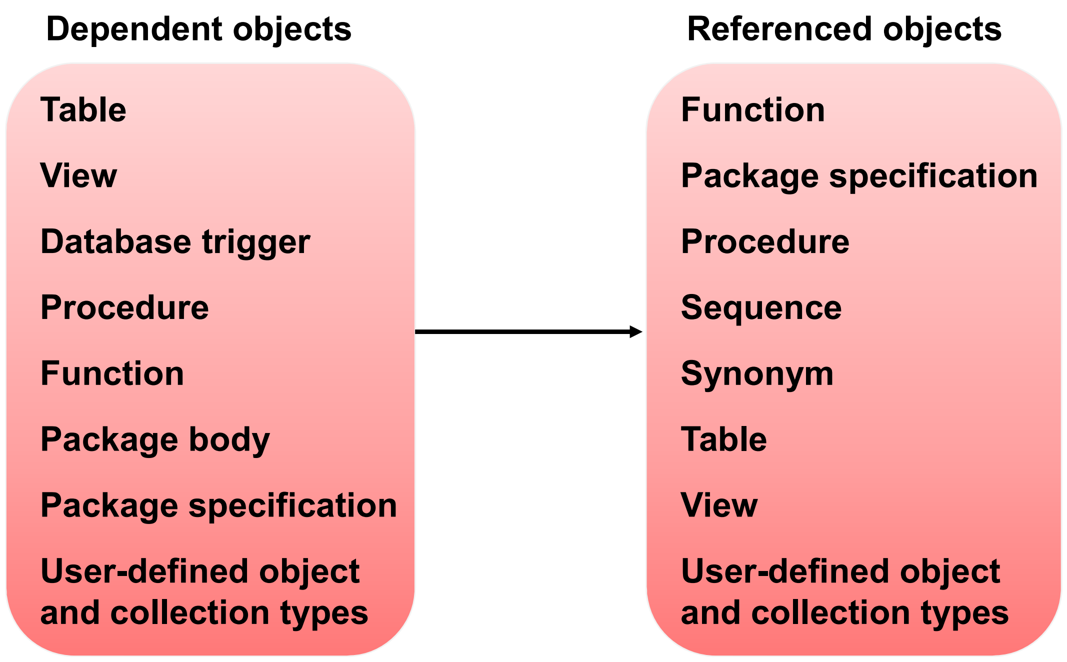
- Is used in two places:
  - Executable section
  - Exception section
- Returns error conditions to the user in a manner consistent with other Oracle server errors

The `RAISE_APPLICATION_ERROR` procedure 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 processes a predefined, non-predefined, or user-defined error. The error number and message are displayed to the user.

Lesson Agenda

- Describing PL/SQL basics
- Listing restrictions on calling functions from SQL expressions
- Reviewing PL/SQL packages
- Identifying how explicit cursors are processed
- Handling exceptions
- Using the `RAISE_APPLICATION_ERROR` procedure
- Managing dependencies
- Using Oracle-supplied packages

Dependencies



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*.

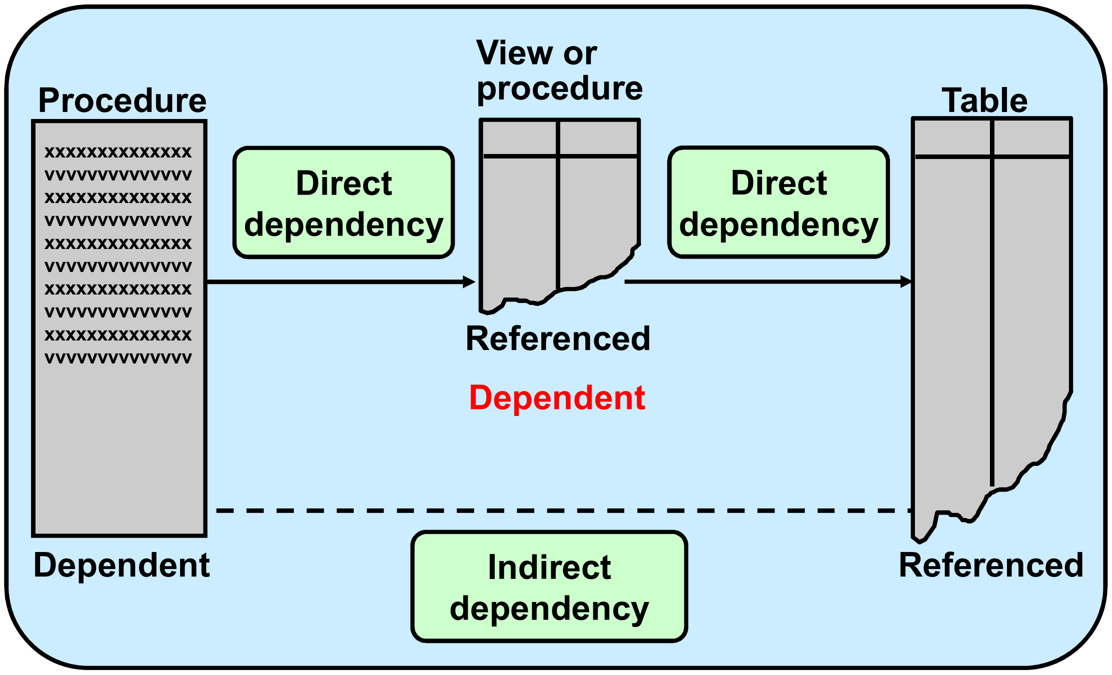
Dependency Issues

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, a 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.

Note: If the status is valid, the schema object is successfully compiled and can be immediately used when referenced. If the status is invalid, the schema object must be compiled before it can be used.

Dependencies



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')
```

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

Steps to Execute the Example

1. Make sure that the `utldtree.sql` script was 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

Lesson Agenda

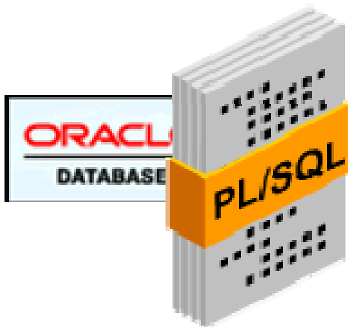
- Describing PL/SQL basics
- Listing restrictions on calling functions from SQL expressions
- Reviewing PL/SQL packages
- Identifying how explicit cursors are processed
- Handling exceptions
- Using the `RAISE_APPLICATION_ERROR` procedure
- Managing dependencies
- Using Oracle-supplied packages

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.



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

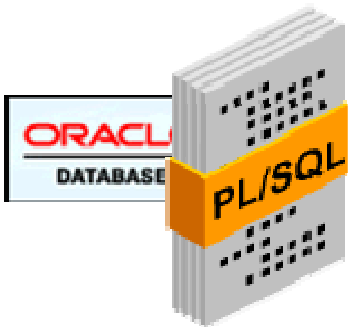
You can use the functionality provided by these packages when creating your application, or you may simply want to use these packages as ideas when you create your own stored procedures.

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**



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 the *PL/SQL Packages and Types Reference*.

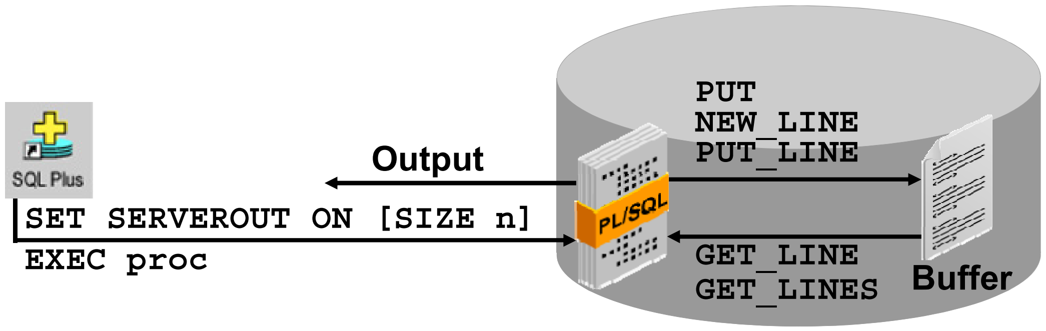
The following is a brief description of some listed packages:

- 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 email 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. (The default is OFF.)



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

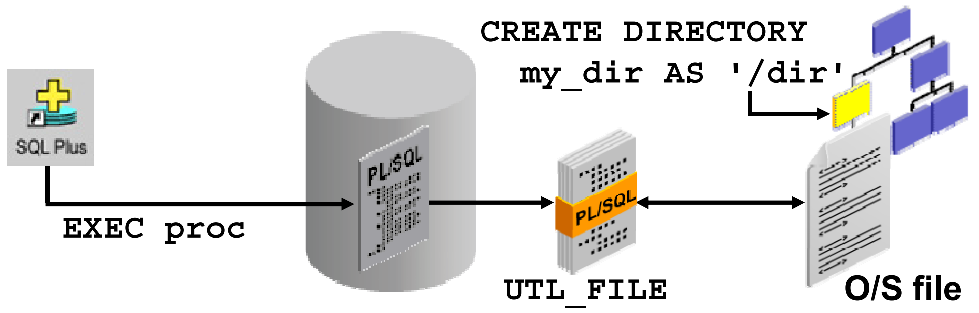
- You can output results to the window for debugging purposes.
- You can trace the code execution path for a function or procedure.
- You can send messages between subprograms and triggers.

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.

- It provides a restricted version of operating system stream file I/O for text files.
- It can access files in operating system directories defined by a CREATE DIRECTORY statement.



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;
```

This approach of using the directory alias created by the CREATE DIRECTORY statement does not require the database to be restarted. The operating system directories specified 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 on calling functions from SQL expressions
- Review PL/SQL packages
- Use cursors
- Handle exceptions
- Use the `RAISE_APPLICATION_ERROR` procedure
- Identify Oracle-supplied packages

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

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

Practice 2: Overview

This practice covers a review of the following topics:

- PL/SQL basics
- Cursor basics
- Exceptions
- Dependencies
- Oracle-Supplied Packages

In this practice, you test and review your PL/SQL knowledge. This knowledge is necessary as a base line for the subsequent chapters to build upon.

3

Designing PL/SQL Code

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 the existing types for an application
- Specify a white list of PL/SQL units to access a package

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

- Design and use cursor variables
- Describe the predefined data types
- Create subtypes based on existing data types for an application
- Specify a white list of PL/SQL units to access a package

Lesson Agenda

- Identifying guidelines for cursor design
- Using cursor variables
- Creating subtypes based on existing types
- Using White Lists

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;
...
```

- Benefits:
  – No individual variables declaration is needed.
  – You can automatically use the structure of the SELECT column list.

Guideline

- When fetching from a cursor, fetch into a record.

Benefits

- You do not need to declare individual variables, and you reference only the values that you want to use.
- You can automatically use the structure of the SELECT column list.

Guidelines for Cursor Design

- Create cursors with parameters.

```
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;
BEGIN
   OPEN cur_cust(p_crd_limit_in, p_acct_mgr_in);
...
   CLOSE cur_cust;
...
   OPEN cur_cust(v_credit_limit, 145);
...
END;
```

- Benefits:
  – Parameters increase the cursor's flexibility and reusability.
  – Parameters help avoid scoping problems.

Guideline

- Whenever you use a cursor in multiple places with different values for the `WHERE` clause, create parameters for your cursor.

Benefits

- 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 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
   ...
```

- Benefit:
  - It ensures that you are dealing with the result of the correct SQL statement.

Guideline

- If you use an implicit cursor and reference a SQL cursor attribute, make sure that you reference it immediately after a SQL statement is executed.

Benefits

- 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 `SQL%NOTFOUND` cursor attribute function must be called immediately after the data manipulation language (DML) statement:

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;
BEGIN
  FOR cur_rec IN cur_cust (p_crd_limit_in, p_acct_mgr_in)
  LOOP         -- implicit open and fetch
  ...
  END LOOP;    -- implicit close
  ...
END;
```

- Benefits:
  - Reduces the volume of code
  - Automatically handles the open, fetch, and close operations, and defines a record type that matches the cursor definition

Benefits (continued)

```
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
…
```

Guideline

- Whenever possible, use cursor `FOR` loops that simplify coding.

Benefits

- Cursor `FOR` loops reduce the volume of code that you must 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, and 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

- If you no longer need a cursor, 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.

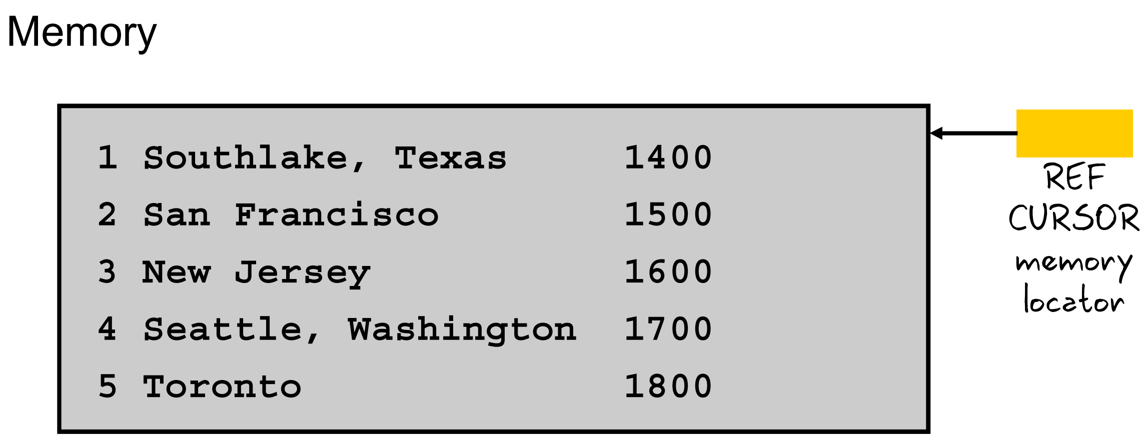
  Note: It is a good programming practice to explicitly close your cursors. Leaving cursors open can generate an exception, because the number of cursors allowed to remain open within a session is limited.
- Make sure that you use column aliases in your cursor for calculated columns that you fetch into a record declared with a `%ROWTYPE` declaration. You would also use 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 `credit_limit*1.1` calculation. After you give it an alias, use the same alias later in the code to make a reference to the calculation.

Lesson Agenda

- Identifying guidelines for cursor design
- Using cursor variables
- Creating subtypes based on existing types
- Using White Lists

Cursor Variables: Overview



Like a cursor, a cursor variable points to the current row in the result set of a multiple-row query. Cursor variables, however, are like C pointers; they hold the memory location of an item instead of the item itself. Thus, 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.

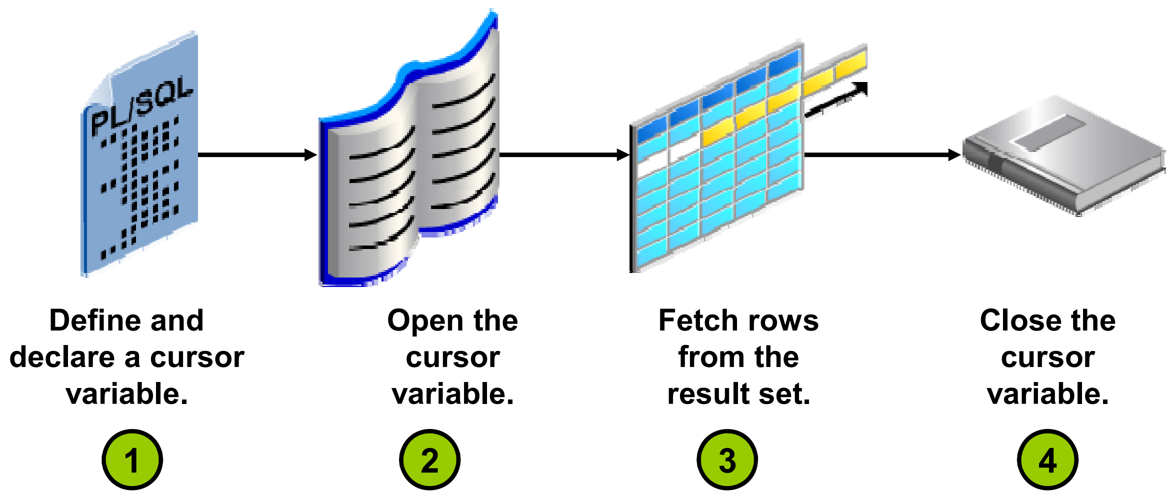
Using Cursor Variables

To execute a multiple-row query, the Oracle server opens a work area called a "cursor" to store the 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. A cursor always refers to the same work area, whereas 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.

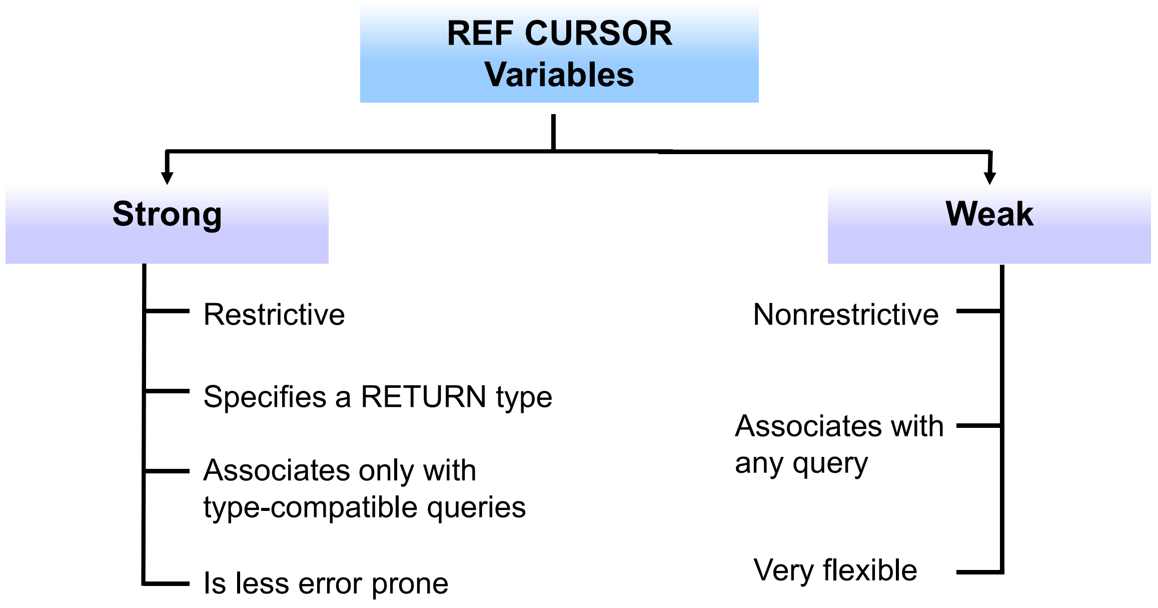
Primarily, you use a cursor variable to pass a pointer to query result sets between PL/SQL-stored subprograms and various clients, such as a Developer Forms application. None of them owns the result set. They simply share a pointer to the query work area that stores the result set.

Working with Cursor Variables


PL/SQL
Define and declare a cursor variable.
1
Open the cursor variable.
2
Fetch rows from the result set.
3
Close the cursor variable.
4

The slide lists the four steps for handling a cursor variable. The next few sections contain detailed information about each step.

Strong Versus Weak `REF 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 only with type-compatible queries, whereas a weak type can be associated with any query. This makes strong `REF CURSOR` types less prone to error, but weak `REF CURSOR` types more flexible.

In the following example, the first definition is strong, whereas the second is 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 specified in subsequent declarations.
- *`return_type`* represents a record type.
- The *`RETURN`* keyword indicates a strong cursor.

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

To create a cursor variable, you first 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`* is 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. `return_type` is a record type indicating the select list types 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;
```

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 `cv_cust` cursor variable.

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.

The following are other 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 multiple-row `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.

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;
/
```

```
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.

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 `INVALID_CURSOR` predefined exception.

The `CLOSE` statement disables a cursor variable, after which the result set is undefined. The syntax is:

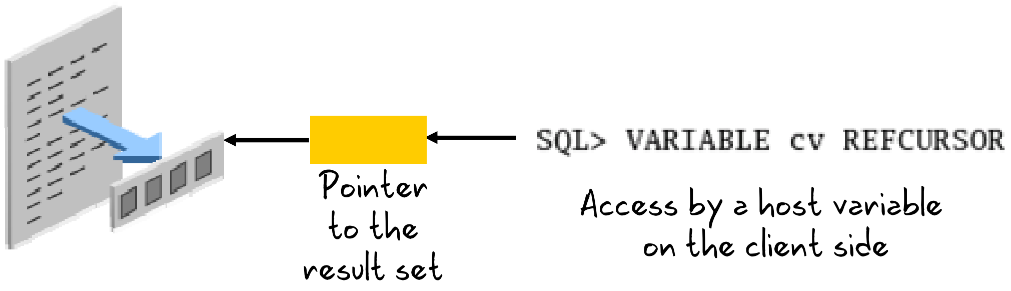
```
        CLOSE cursor_variable_name;
```

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.


SQL> VARIABLE cv REFCURSOR
Pointer to the result set
Access by a host variable on the client side

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 multiple-block form.

Example

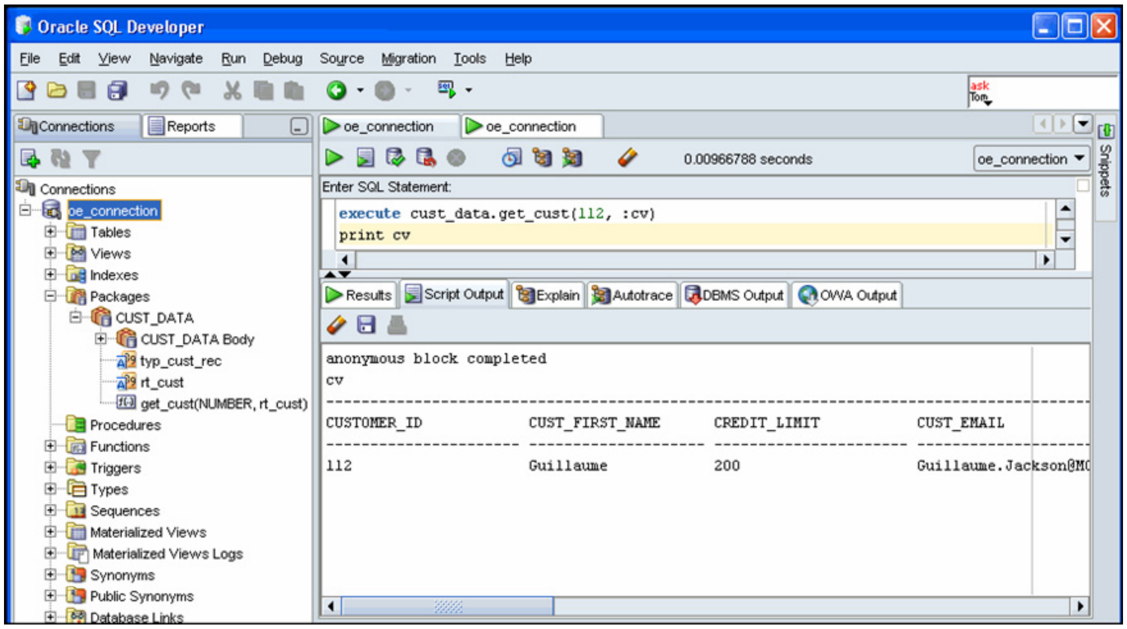
Using SQL*Plus, define a host variable with a data type of `REFCURSOR` to hold the query results generated from a `REF CURSOR` in a stored subprogram. Use the SQL*Plus `PRINT` command to view the host variable results. Optionally, you can set the SQL*Plus command `SET AUTOPRINT ON` to display the query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

Note: You can define a host variable in SQL*Plus or SQL Developer. This slide uses SQL*Plus. The next slide shows the use of SQL Developer.

Passing Cursor Variables as Arguments



```
execute cust_data.get_cust(112, :cv)
print cv
```

```
anonymous block completed
cv
------------------------------------------------------------------------------------------------
CUSTOMER_ID           CUST_FIRST_NAME       CREDIT_LIMIT          CUST_EMAIL
--------------------- --------------------- --------------------- ---------------------
112                   Guillaume             200                   Guillaume.Jackson@MC
```

```
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;
/
```

```
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.

Using the `SYS_REFCURSOR` Predefined Type

```
CREATE OR REPLACE PROCEDURE REFCUR
(p_num IN NUMBER)
IS
refcur sys_refcursor;
empno           emp.empno%TYPE;
ename           emp.ename%TYPE;
BEGIN
IF p_num = 1 THEN
    OPEN refcur FOR SELECT empno, ename FROM emp;
    DBMS_OUTPUT.PUT_LINE('Employee#    Name');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    LOOP
        FETCH refcur INTO empno, ename;
        EXIT WHEN refcur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(empno || '        ' || ename);
    END LOOP;
ELSE
....
```

SYS_REFCURSOR is a built-in REF CURSOR type that allows any result set to be associated with it.

You can define a cursor variable by using the built-in `SYS_REFCURSOR` data type as well as by creating a `REF CURSOR` type, and then declaring a variable of that type. `SYS_REFCURSOR` is a `REF CURSOR` type that allows any result set to be associated with it. As mentioned earlier, this is known as a *weak* (nonrestrictive) `REF CURSOR`.

`SYS_REFCURSOR` can be used to:

- Declare a cursor variable in an Oracle-stored procedure or function
- Pass cursors from and to an Oracle-stored procedure or function

Note: *Strong* (restrictive) `REF CURSORS` require the result set to conform to a declared number and order of fields with compatible data types, and can also, optionally, return a result set.

```
CREATE OR REPLACE PROCEDURE REFCUR
(p_num IN NUMBER)
IS
refcur sys_refcursor;
empno          emp.empno%TYPE;
ename          emp.ename%TYPE;
BEGIN
```

```
IF p_num = 1 THEN
    OPEN refcur FOR SELECT empno, ename FROM emp;
    DBMS_OUTPUT.PUT_LINE('Employee#    Name');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    LOOP
        FETCH refcur INTO empno, ename;
        EXIT WHEN refcur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(empno || '         ' || ename);
    END LOOP;
ELSE
OPEN refcur FOR
  SELECT empno, ename
  FROM emp WHERE deptno = 30;
    DBMS_OUTPUT.PUT_LINE('Employee#    Name');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    LOOP
        FETCH refcur INTO empno, ename;
        EXIT WHEN refcur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(empno || '         ' || ename);
    END LOOP;
END IF;
    CLOSE refcur;
END;
/
```

Dynamic SQL for Cursor Variable

```
DROP TABLE EMP2;
CREATE TABLE EMP2 AS SELECT * FROM EMPLOYEES WHERE
DEPARTMENT_ID=50;
CREATE or replace PROCEDURE WORKERS(TNAME VARCHAR2,SAL NUMBER)
IS
TYPE C_EMP IS REF CURSOR;
C C_EMP;
R EMPLOYEES%ROWTYPE;
BEGIN
OPEN C FOR 'SELECT * FROM '||TNAME||' WHERE SALARY>:f ' using
SAL;
LOOP
FETCH C INTO R;
EXIT WHEN C%NOTFOUND;
DBMS_OUTPUT.PUT_LINE(R.LAST_NAME||' SALARY:'||R.SALARY);
END LOOP;
CLOSE C;
END WORKERS;
/
```

The code example in this slide shows the usage of dynamic SQL in cursors.

Rules for Cursor Variables

- You cannot:
  - Use cursor variables with remote subprograms on another server
  - Use comparison operators to test cursor variables
  - Assign a NULL value to cursor variables
  - Use `REF CURSOR` types in `CREATE TABLE` or `VIEW` statements
- Cursors and cursor variables are not interoperable.

- 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.
- 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 the 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 areas after a block completes

Cursor variables are dynamic and provide wider flexibility. Unlike static cursors, cursor variables are not tied to a single `SELECT` statement. In applications where `SELECT` statements may differ depending on various situations, the cursor variables can be opened for each of the `SELECT` statements. 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:

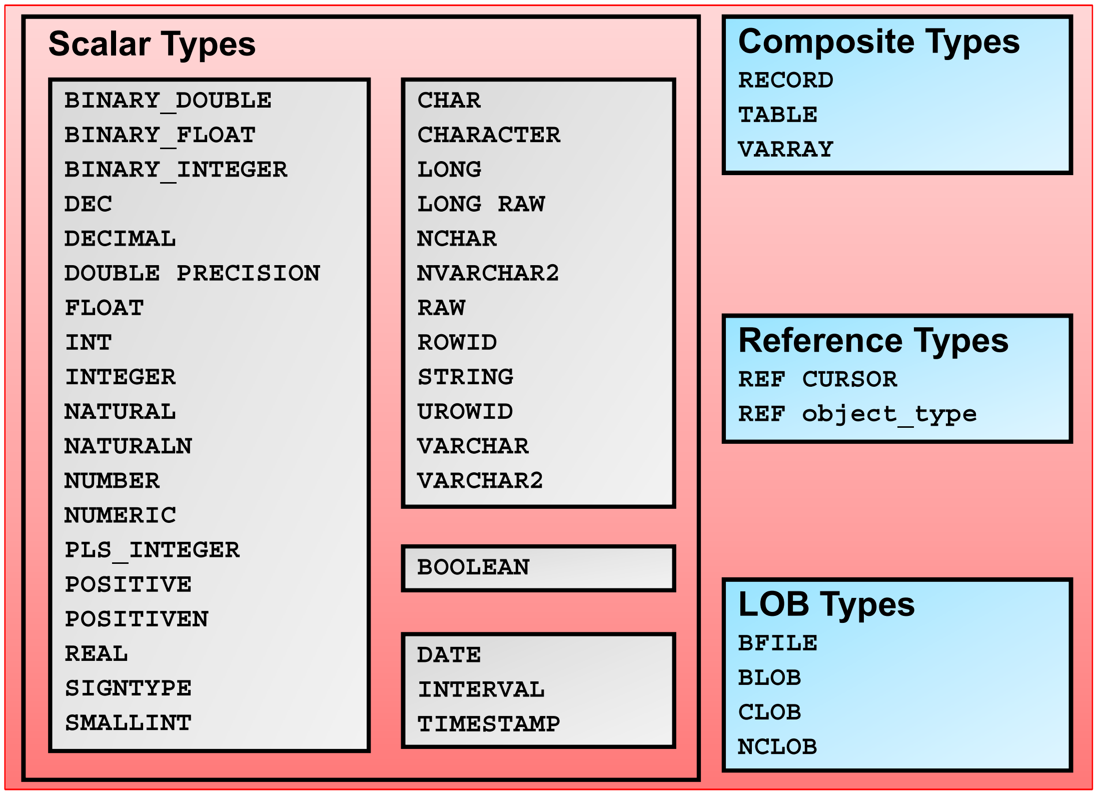
```
/* anonymous PL/SQL block in host environment */
BEGIN
   OPEN :cv_cust FOR SELECT * FROM customers;
   OPEN :cv_orders FOR SELECT * FROM orders;
END;
```

This may be useful in Oracle Forms, for instance, when you want to populate a multiple-block 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. This enables your OCI or Pro*C program to use these work areas for ordinary cursor operations.

Lesson Agenda

- Identifying guidelines for cursor design
- Using Cursor Variables
- Creating subtypes based on existing types
- Using White Lists

Predefined PL/SQL Data Types



Scalar Types

```
BINARY_DOUBLE
BINARY_FLOAT
BINARY_INTEGER
DEC
DECIMAL
DOUBLE PRECISION
FLOAT
INT
INTEGER
NATURAL
NATURALN
NUMBER
NUMERIC
PLS_INTEGER
POSITIVE
POSITIVEN
REAL
SIGNTYPE
SMALLINT
```

```
CHAR
CHARACTER
LONG
LONG RAW
NCHAR
NVARCHAR2
RAW
ROWID
STRING
UROWID
VARCHAR
VARCHAR2
```

```
BOOLEAN
```

```
DATE
INTERVAL
TIMESTAMP
```

Composite Types

```
RECORD
TABLE
VARRAY
```

Reference Types

```
REF CURSOR
REF object_type
```

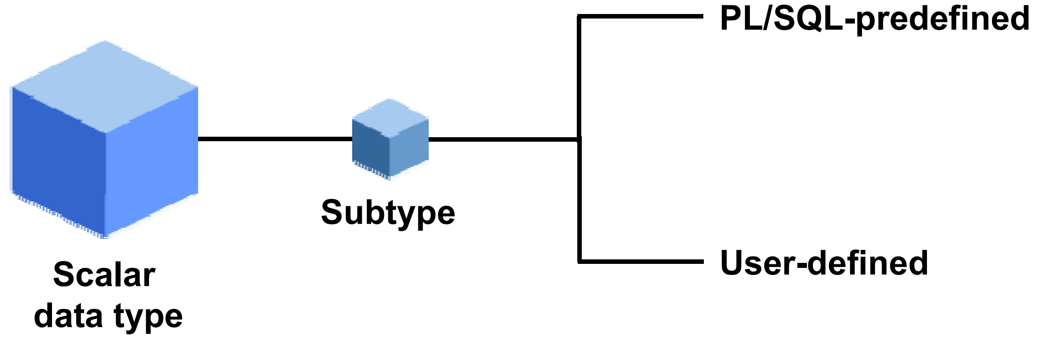
LOB Types

```
BFILE
BLOB
CLOB
NCLOB
```

Every constant, variable, and parameter has a data type, which specifies a storage format, a valid range of values, and constraints. 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 subtypes.

Subtypes: Overview

A subtype is a subset of an existing data type that may place a constraint on its base type.



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.

Standard Subtypes

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

Several predefined subsets are specified in the standard package. DECIMAL and INTEGER are subtypes of NUMBER. CHARACTER is a subtype of CHAR.

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

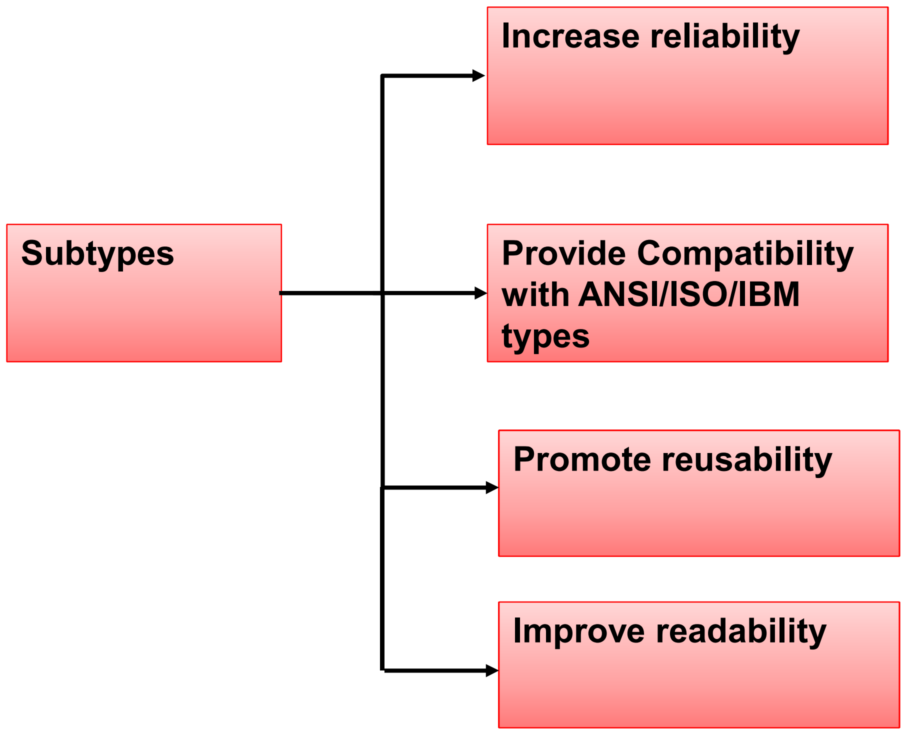
Use the DOUBLE PRECISION and FLOAT subtypes to declare floating-point numbers with a maximum precision of 126 binary digits, which is roughly equivalent to 38 decimal digits. Or, use the subtype REAL to declare floating-point numbers with a maximum precision of 63 binary digits, which is roughly equivalent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

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

You can even 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



If your applications require a subset of an existing data type, you can create subtypes.

Using subtypes, you can:

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

Subtypes can increase reliability by detecting the out-of-range values.

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

Declaring Subtypes

- Subtypes are defined in the declarative section of a 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.

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. 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 a variable 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.

After a 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 (4,2);
  v_amount  accumulator;
  v_total   NUMBER;
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;
/
```

Some applications require constraining subtypes to a size specification for scientific purposes. The example in the slide shows that if you exceed the size of your subtype, you receive an error.

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.

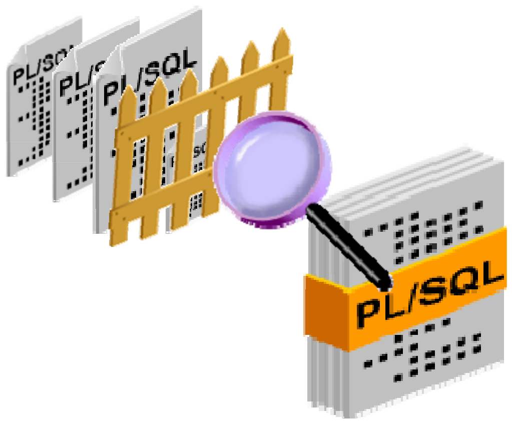
```
DECLARE
   v_rows      mytypes.Counter;  v_customers
               mytypes.Counter;
   SUBTYPE     Accumulator IS NUMBER (6,2);
   v_total     NUMBER;
BEGIN
   SELECT COUNT(*) INTO v_customers FROM customers;
   SELECT COUNT(*) INTO v_rows FROM orders;
   v_total := v_customers + v_rows;
   DBMS_OUTPUT.PUT_LINE('Total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

Lesson Agenda

- Identifying guidelines for cursor design
- Using Cursor Variables
- Creating subtypes based on existing types
- Using White Lists

Declaring and Defining White Lists

- Use the White List feature to restrict access to program units.
- Use the ACCESSIBLE BY clause to enable a list of PL/SQL program units to access the PL/SQL unit you create.



The ACCESSIBLE BY clause of the package specification allows you to specify a "white list" of PL/SQL units that can access the subprogram. Some examples of situations where you can use this clause are when:

- You implement a PL/SQL application that contains a package that provides the application programming interface (API) and helper packages to do the work. You want clients to have an access to the API, but not to the helper packages. In this case, you omit the ACCESSIBLE BY clause from the API package specification and include it in each helper package specification, where you specify that only the API package can access the helper package.
- You create a utility package to provide services to some, but not all, PL/SQL units in the same schema. To restrict the use of the package to the intended units, you list them in the ACCESSIBLE BY clause in the package specification.

The ACCESSIBLE BY clause specifies each accessor (PL/SQL unit) that can invoke the function. An accessor can appear more than once in the ACCESSIBLE BY clause, but the ACCESSIBLE BY clause can appear only once in the function.

- You can use this clause in conjunction with an AUTHID clause in any order.
- You can use the White List feature to implement the concept of least privileges in your database.

Using the `ACCESSIBLE BY` Clause in the `CREATE PROCEDURE` Statement

Syntax:

```
CREATE [OR REPLACE] PROCEDURE procedure_name
 [(parameter1 [mode] datatype1,
   parameter2 [mode] datatype2, ...)]
AUTHID [CURRENT USER| DEFINER]
ACCESSIBLE BY (accessor)
IS|AS
  [local_variable_declarations; ...]
BEGIN
      -- actions;
END [procedure_name];
```

- where:
  - `accessor := [accessor_ kind] accessor_ name`
  - `accessor_kind := [PROCEDURE | FUNCTION | PACKAGE | TRIGGER | TYPE]`

This slide shows the syntax of using the `ACCESSIBLE BY` clause in the `CREATE PROCEDURE` statement.

Note: The compiler does not check the existence of the `accessor`.

Using the `ACCESSIBLE BY` Clause in the `CREATE FUNCTION` Statement

Syntax:

```
CREATE [OR REPLACE] FUNCTION function_name
 [(parameter1 [mode] datatype1,
   parameter2 [mode] datatype2, ...)]
AUTHID [CURRENT_USER| DEFINER]
ACCESSIBLE BY (accessor)
IS|AS
  [local_variable_declarations; ...]
BEGIN
      -- actions;
RETURN expression
END [function_name];
```

- where:
  - `accessor := [accessor_ kind] accessor_ name`
  - `accessor_kind := [PROCEDURE | FUNCTION | PACKAGE | TRIGGER | TYPE]`

This slide shows the syntax of using the `ACCESSIBLE BY` clause in the `CREATE FUNCTION` statement.

Using the `ACCESSIBLE BY` Clause in the `CREATE PACKAGE` Statement

Syntax:

```
CREATE [OR REPLACE] PACKAGE package_name
AUTHID [CURRENT_USER| DEFINER]
ACCESSIBLE BY (accessor)
IS|AS
    public type and variable declarations
    subprogram specifications
END [package_name];
```

- where:
  - `accessor := [accessor_ kind] accessor_ name`
  - `accessor_kind := [PROCEDURE | FUNCTION | PACKAGE | TRIGGER | TYPE]`
  - `public type and variable declarations` declare public variables, constants, cursors, and exceptions
  - `subprogram specifications` specify the public procedure or function declarations

This slide shows the syntax of using the `ACCESSIBLE BY` clause in the `CREATE PACKAGE` statement.

Using the `ACCESSIBLE BY` Clause in the `CREATE TYPE` Statement

Syntax:

```
CREATE [OR REPLACE] [ EDITIONABLE | NONEDITIONABLE ]
TYPE plsql_type_source [ schema. ] type_name [ FORCE ]
[[ AUTHID { CURRENT_USER | DEFINER } ]|
[ ACCESSIBLE BY ( accessor [, accessor ]... ) ]]
[object_type |  { IS | AS }
 { varray_type_def | nested_table_type_def } ];
```

- The `CREATE TYPE` statement creates or replaces the specification of one of the following:
  - Abstract Data Type (ADT) (including a SQLJ object type)
  - Stand-alone stored varying array (varray) type
  - Stand-alone stored nested table type
  - Incomplete object type

This slide shows the syntax of using the `ACCESSIBLE BY` clause in the `CREATE TYPE` statement.

Using the `ACCESSIBLE BY` Clause

Example:

```
CREATE OR REPLACE PACKAGE private_access
   ACCESSIBLE BY (all_access)
IS
   PROCEDURE a_permit;
   PROCEDURE b_permit;
END;
```

```
CREATE OR REPLACE PACKAGE BODY all_access
IS
   PROCEDURE proc_all_access
   IS
   BEGIN
      private_access.a_permit;
      private_access.b_permit;
   END;
END;
```

In the example shown in the slide:

- When you execute the `all_access` package, you get the desired output
- When you execute the `private_access` package separately, it results in error

Note: Watch the demonstration to understand how this example works.

Quiz

Which of the following are true?

a. Fetching into multiple variables when fetching from a cursor gives you the advantage of automatic usage of the structure of the `SELECT` column list.

b. Creating cursors with parameters helps in avoiding scoping problems.

c. Close a cursor when it is no longer needed.

d. All of the above

Answer: b, c

Quiz

Strong `REF CURSOR`:

a. Is nonrestrictive
b. Specifies a `RETURN` type
c. Associates only with type-compatible queries
d. Is less error prone

Answer: b, c, d

Quiz

A subtype is a subset of an existing data type that may place a constraint on its base type.

a. True

b. False

Answer: a

Quiz

Using white lists, you can protect a range of units from being invoked by unauthorized person.

a. True

b. False

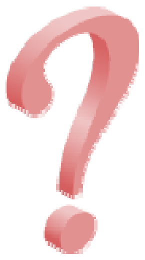
Answer: a

Quiz

The `ACCESSIBLE BY` clause is synonymous to using subfunctions.

a. True

b. False



Answer: a

The `ACCESSIBLE BY` clause enables you to specify a list of units that may view and execute a given unit. It is more flexible than using subfunctions.

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
- Specify a white list of PL/SQL units to access a package

ORACLE

- 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.
- You can use the `ACCESSIBLE BY` clause to define a white list in a package.

Practice 3: 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
- Using the `ACCESSIBLE BY` clause

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

4 Overview of Collections

Objectives

After completing this lesson, you should be able to create collections:

- Nested tables
- Varrays
- Associative arrays/PLSQL tables

In this lesson, you are introduced to PL/SQL programming using collections.

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.

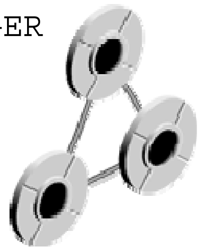
Collections work like the set, queue, stack, and hash table data structures found in most third-generation programming languages. Collections can store instances of an object type and can also be attributes of an object type. Collections can be passed as parameters.

Lesson Agenda

- Understanding collections
- Using associative arrays
- Using nested tables
- Using varrays

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:
  - Associative arrays
    - String-indexed collections
    - `INDEX BY PLS_INTEGER` or `BINARY_INTEGER`
  - Nested tables
  - Varrays



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.

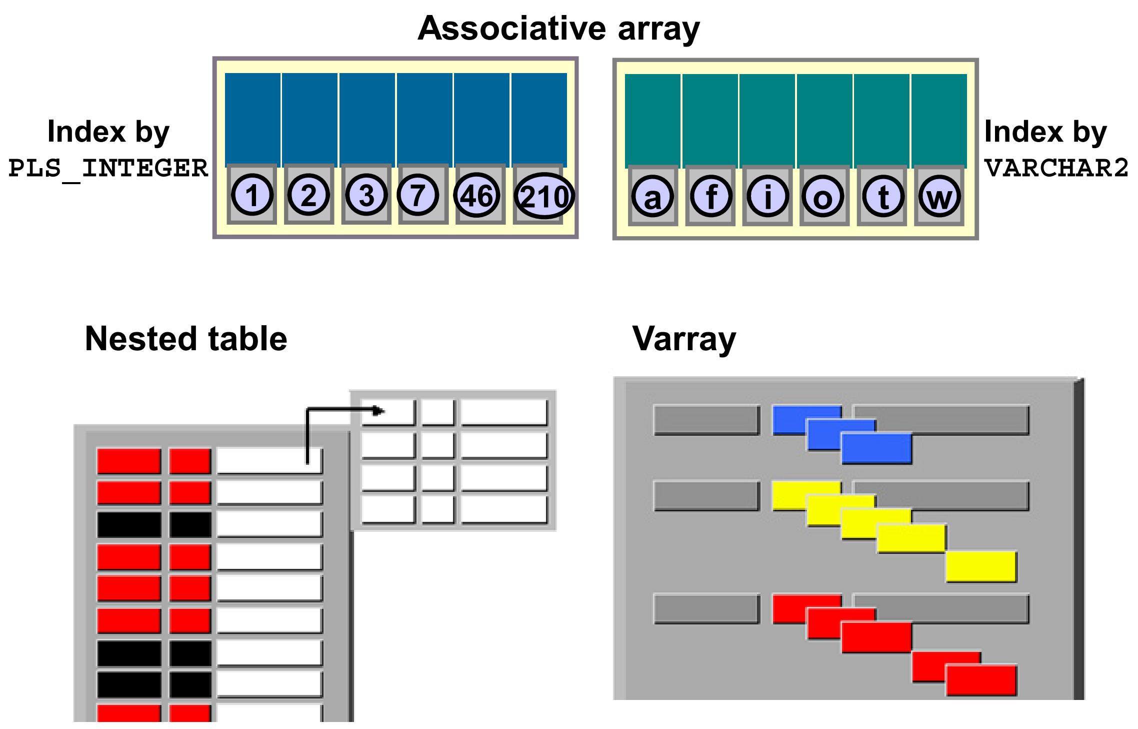
Object types are used not only to create object relational tables, but also to define collections.

You can use any of the three categories of collections:

- 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.
- Nested tables can have any number of elements.
- A varray is an ordered collection of elements.

Note: Associative arrays indexed by `PLS_INTEGER` are covered in the prerequisite courses—*Oracle Database 12c: Program with PL/SQL* and *Oracle Database 12c: Develop PL/SQL Program Units*—and are not emphasized in this course.

Collection Types



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 based (`PLS_INTEGER` or `BINARY_INTEGER`) or character based (`VARCHAR2`). Associative arrays may be sparse.

When you assign a value by 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. However, it is important to choose a key that is unique. For example, the 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, 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. They are typically populated with a `SELECT BULK COLLECT` statement unless they are `VARCHAR2` indexed. `BULK COLLECT` prevents context switching between the SQL and PL/SQL engines, and is much more efficient on large data sets.

Nested Tables

A nested table holds a set of values. In other words, it is a table within a table. Nested tables are unbounded; that is, the size of the table can increase dynamically. Nested tables are available in both PL/SQL and 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 an array-like access to individual rows. Nested tables are initially dense, but they can become sparse through deletions and, therefore, have nonconsecutive subscripts.

Note: The graphical interpretation of the nested table shows the SQL implementation, and it is stored in data dictionary.

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, thereby 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.

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.

Choosing a PL/SQL Collection Type

If you already have code or business logic that uses another language, you can usually translate that language’s array and set the types directly to the 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.

Why Use Collections?

Collections offer object-oriented features such as variable-length arrays and nested tables that provide higher-level ways to organize and access data in the database. Below the object layer, data is still stored in columns and tables, but you are able to work with the data in terms of the real-world entities, such as customers and purchase orders, that make the data meaningful.

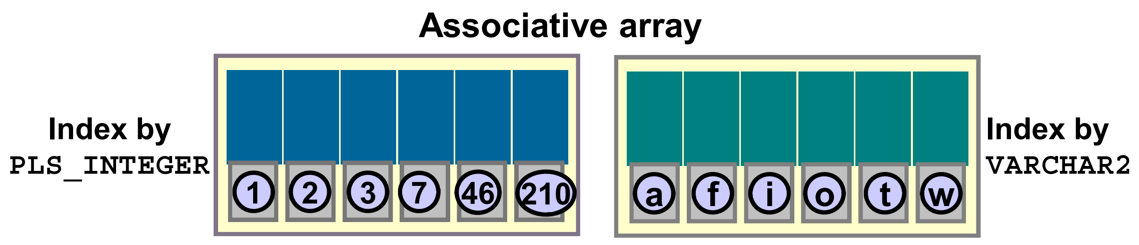
Lesson Agenda

- Understanding collections
- Using associative arrays
- Using nested tables
- Using varrays

Using Associative Arrays

Associative arrays:

- That are indexed by strings or integers can improve performance
- Are pure memory structures that are much faster than schema-level tables
- Provide significant additional flexibility



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.

When to Use String-Indexed Arrays

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 nonnumeric key, where the notion of sparseness is not applicable. In contrast, the `INDEX BY PLS_INTEGER` tables are optimized for compactness of storage on the assumption that the data is dense.

If you 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.

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

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

Creating the Array

Associative array in PL/SQL (string-indexed):

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

```
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(50);

  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;
...
```

Create the string-indexed associative array type.

Create the string-indexed associative array variable.

Populate the string-indexed associative array variable.

In the `REPORT_CREDIT` procedure shown in the slide, you 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 email name is reported in case more than one customer has the same last name.

Traversing the Array

```
...
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: 4000
Customer Walken: Prem.Walken@BRANT.COM has credit limit of: 4100
```

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 email. Then you can immediately access the information based on the customer email key. You would pass the email name instead of the customer last name.

Here is the modified code:

```
CREATE OR REPLACE PROCEDURE report_credit
  (p_email    customers.cust_email%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(50);

  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.EXAMPLE.COM', 100)

For credit amount of: 100
Customer Walken: Prem.Walken@BRANT.EXAMPLE.COM has credit
 limit of: 4100

PL/SQL procedure successfully completed.
```

Lesson Agenda

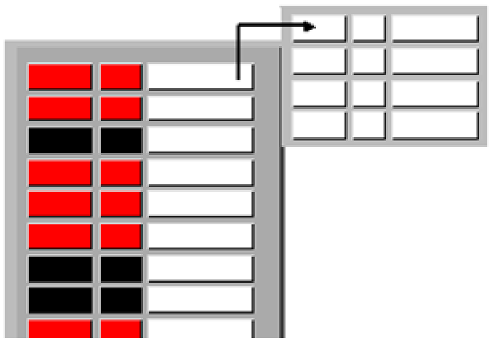
- Understanding collections
- Using associative arrays
- Using nested tables
- Using varrays

Using Nested Tables

Nested table characteristics:

- Unbounded
- Available in both SQL and PL/SQL as well as the database
- Array-like access to individual rows
- Sparse

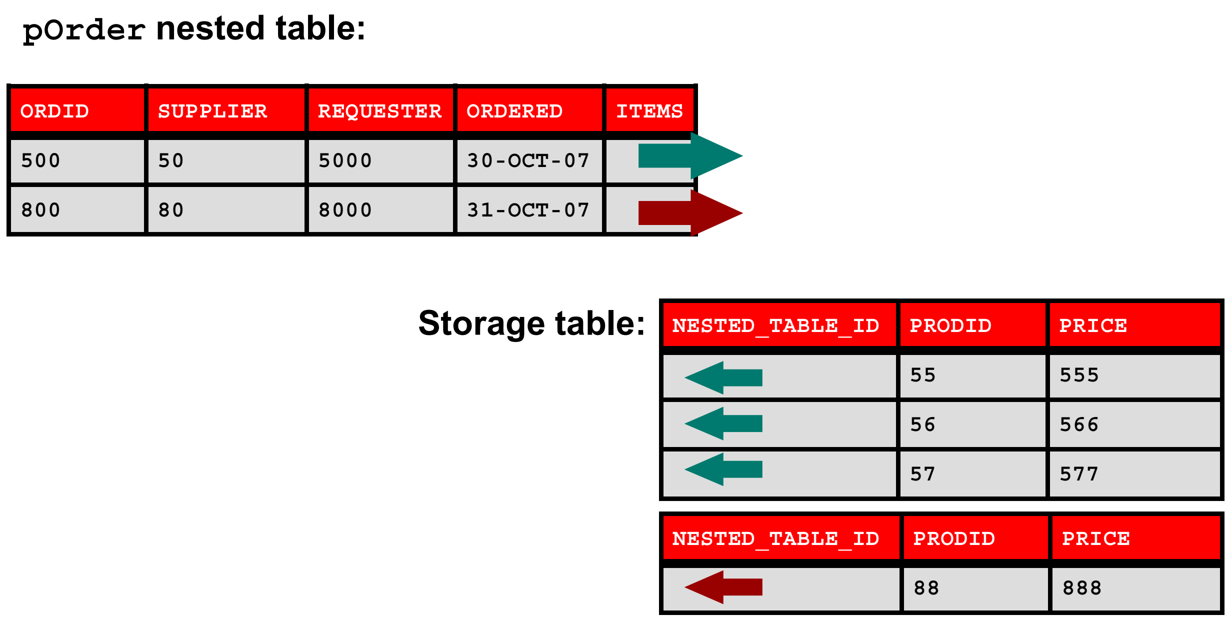
Nested table:



A nested table holds a set of values. Nested tables are unbounded, which means that 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 an array-like access to individual rows. Nested tables are initially dense, but they can become sparse through deletions and, therefore, have nonconsecutive subscripts.

Nested Table Storage

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



`pOrder` **nested table:**

ORDID	SUPPLIER	REQUESTER	ORDERED	ITEMS
500	50	5000	30-OCT-07	→
800	80	8000	31-OCT-07	→

Storage table:

NESTED_TABLE_ID	PRODID	PRICE
←	55	555
←	56	566
←	57	577

NESTED_TABLE_ID	PRODID	PRICE
←	88	888

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 a 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.

The column privileges of the parent table are transferred to the nested table.

Creating Nested Table Types

- To create a nested table type in the database:

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

- To create a nested table type in PL/SQL:

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

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

Creating Collections in the Database

You can create a nested table 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, 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 in PL/SQL. Use the syntax shown in the slide to create collection types in PL/SQL.

Note: Collections can be nested. Collections of collections are also possible.

Declaring Collections: Nested Table

- First, define an object type:

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

- Then, declare a column of that collection type:

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

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.

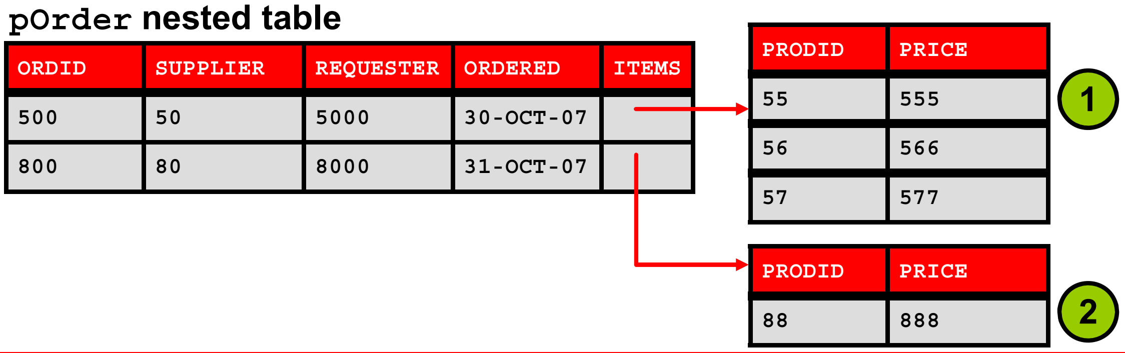
   Note: You must create the `typ_item_nst` nested table type based on the previously declared type, because it is illegal to declare multiple data types in this nested table declaration.
3. Create the `pOrder` table and use the nested table type in a column declaration, which includes an arbitrary number 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 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 `USER_COLL_TYPES` dictionary view holds information about collections.

Using Nested Tables

- Add data to the nested table:

```
INSERT INTO pOrder
  VALUES (500, 50, 5000, sysdate, typ_item_nst(
      typ_item(55, 555),
      typ_item(56, 566),
      typ_item(57, 577)));

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



ORDID	SUPPLIER	REQUESTER	ORDERED	ITEMS
500	50	5000	30-OCT-07	
800	80	8000	31-OCT-07	

PRODID	PRICE
55	555
56	566
57	577

PRODID	PRICE
88	888

To insert data into the nested table, you use the INSERT statement. A constructor is a system-defined function that is used to identify where the data should be placed, essentially “constructing” the collection from the elements passed to it.

In the example in the slide, the constructors are TYP_ITEM_NST() and TYP_ITEM(). You pass two elements to the TYP_ITEM() constructor, and then pass the results to the TYP_ITEM_NST() constructor to build the nested table structure.

The first INSERT statement builds the nested table with three subelement rows.

The second INSERT statement builds the nested table with one subelement row.

Using Nested Tables

- Querying the results:

```
SELECT * FROM porder;

     ORDID   SUPPLIER  REQUESTER ORDERED
---------- ---------- ---------- ---------
ITEMS(PRODID, PRICE)
-----------------------------------------------------------------
       500         50       5000 31-OCT-07
TYP_ITEM_NST(TYP_ITEM(55, 555), TYP_ITEM(56, 566), TYP_ITEM(57, 577))
       800         80       8000 31-OCT-07
TYP_ITEM_NST(TYP_ITEM(88, 888))
```

- Querying the results with the `TABLE` function:

```
SELECT p2.ordid, p1.*
FROM porder p2, TABLE(p2.items) p1;

     ORDID     PRODID      PRICE
---------- ---------- ----------
       800         88        888
       500         57        577
       500         55        555
       500         56        566
```

You can use two general methods to query a table that contains a column or an attribute of a collection type. One method 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.

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.

Referencing Collection Elements

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

- Syntax:

```
collection_name(subscript)
```

- Example:

```
v_with_discount(i)
```

- To reference a field in a collection:

```
p_new_items(i).prodid
```

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 through 2147483647. For varrays, the integer must lie in the range 1 through maximum_size.

Using Nested Tables in PL/SQL

```
CREATE OR REPLACE PROCEDURE add_order_items
(p_ordid NUMBER, p_new_items typ_item_nst)
IS
  v_num_items      NUMBER;
  v_with_discount  typ_item_nst;
BEGIN
  v_num_items := p_new_items.COUNT;
  v_with_discount := p_new_items;
  IF v_num_items > 2 THEN
  --ordering more than 2 items gives a 5% discount
    FOR i IN 1..v_num_items LOOP
      v_with_discount(i) :=
      typ_item(p_new_items(i).prodid,
               p_new_items(i).price*.95);
    END LOOP;
  END IF;
  UPDATE pOrder
    SET  items = v_with_discount
    WHERE ordid = p_ordid;
END;
```

When you define a variable of a collection type in a PL/SQL block, it is transient and available only for the scope of the PL/SQL block.

In the example shown in the slide:

- The nested table `P_NEW_ITEMS` parameter is passed into the block.
- A local variable `V_WITH_DISCOUNT` is defined with the nested table data type `TYP_ITEM_NST`.
- A collection method, called `COUNT`, is used to determine the number of items in the nested table.
- If more than two items are counted in the collection, the local nested table variable `V_WITH_DISCOUNT` is updated with the product ID and a 5% discount on the price.
- To reference an element in the collection, the subscript `i`, representing an integer from the current loop iteration, is used with the constructor method to identify the row of the nested table.

Using Nested Tables in PL/SQL

```
-- caller pgm:
DECLARE
  v_form_items  typ_item_nst:= typ_item_nst();
BEGIN
  -- let's say the form holds 4 items
  v_form_items.EXTEND(4);
  v_form_items(1) := typ_item(1804, 65);
  v_form_items(2) := typ_item(3172, 42);
  v_form_items(3) := typ_item(3337, 800);
  v_form_items(4) := typ_item(2144, 14);
  add_order_items(800, v_form_items);
END;
```

`v_form_items` **variable**

PRODID	PRICE
1804	65
3172	42
3337	800
2144	14

Resulting data in the `pOrder` **nested table**

ORDID	SUPPLIER	REQUESTER	ORDERED	ITEMS
500	50	5000	30-OCT-07	
800	80	8000	31-OCT-07	→

PRODID	PRICE
1804	61.75
3172	39.9
3337	760
2144	13.3

The prices are added after discounts.

ORACLE

In the example code shown in the slide:

- A local PL/SQL variable of nested table type is declared and instantiated with the `TYP_ITEM_NST()` collection method
- The nested table variable is extended to hold four rows of elements with the `EXTEND(4)` method.
- The nested table variable is populated with four rows of elements by constructing a row of the nested table with the `TYP_ITEM` constructor.
- The nested table variable is passed as a parameter to the `ADD_ORDER_ITEMS` procedure shown on the previous page.
- The `ADD_ORDER_ITEMS` procedure updates the `ITEMS` nested table column in the `pOrder` table with the contents of the nested table parameter passed into the routine.

Lesson Agenda

- Understanding collections
- Using associative arrays
- Using nested tables
- Using varrays

Varrays

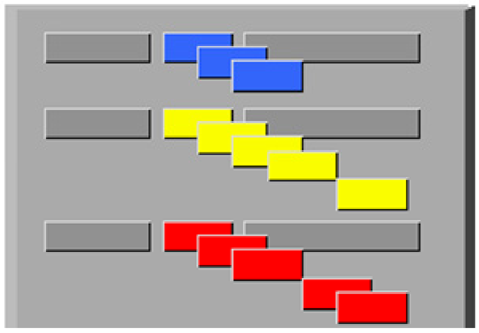
- To create a varray in the database:

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

- To create a varray in PL/SQL:

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

Varray:



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 varrays as a SQL type, thereby 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 individual elements for array operations, or manipulate the collection as a whole.

You can define varrays in PL/SQL to be used during PL/SQL program execution.

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.

Declaring Collections: Varray

- First, define a collection type:

```
CREATE TYPE typ_Project AS OBJECT(   --create object          1
   project_no NUMBER(4),
   title      VARCHAR2(35),
   cost       NUMBER(12,2))
/
CREATE TYPE typ_ProjectList AS VARRAY (50) OF typ_Project
         -- define VARRAY type                                 2
/
```

- Then, declare a collection of that type:

```
CREATE TABLE department (  -- create database table          3
   dept_id  NUMBER(2),
   name     VARCHAR2(25),
   budget   NUMBER(12,2),
   projects typ_ProjectList)  -- declare varray as column
/
```

The example in the slide shows how to create a table based on a varray.

1. Create the TYP_PROJECT type, which holds the 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, and 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
...
);
```

Using Varrays

Add data to the table containing a varray column:

```
INSERT INTO department
  VALUES (10, 'Executive Administration', 30000000,
     typ_ProjectList(
     typ_Project(1001, 'Travel Monitor',  400000),
     typ_Project(1002, 'Open World',    10000000)));

INSERT INTO department
  VALUES (20, 'Information Technology', 5000000,
     typ_ProjectList(
     typ_Project(2001, 'DB11gR2', 900000)));
```

1

2

DEPARTMENT table

DEPT_ID	NAME	BUDGET	PROJECTS			
			PROJECT_NO	TITLE	COSTS	
10	Executive Administration	30000000	1001	Travel Monitor	400000	1
			1002	Open World	10000000	
20	Information Technology	5000000	2001	DB11gR2	900000	2

To add rows to the DEPARTMENT table that contains the PROJECTS varray column, you use the INSERT statement. The structure of the varray column is identified with the constructor methods.

- TYP_PROJECTLIST() constructor constructs the varray data type.
- TYP_PROJECT() constructs the elements for the rows of the varray data type.

The first INSERT statement adds three rows to the PROJECTS varray for department 10.

The second INSERT statement adds one row to the PROJECTS varray for department 20.

Using Varrays

- Querying the results:

```
SELECT * FROM department;

   DEPT_ID NAME                            BUDGET
---------- ------------------------- ----------
PROJECTS(PROJECT_NO, TITLE, COST)
-----------------------------------------------------------------
        10 Executive Administration    30000000
TYP_PROJECTLIST(TYP_PROJECT(1001, 'Travel Monitor', 400000),
TYP_PROJECT(1002, 'Open World', 10000000))
        20 Information Technology       5000000
TYP_PROJECTLIST(TYP_PROJECT(2001, 'DB11gR2', 900000))
```

- Querying the results with the `TABLE` function:

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

DEPT_ID NAME                      PROJECT_NO TITLE              COST
------- ------------------------ ---------- --------------  --------
     10 Executive Administration       1001 Travel Monitor    400000
     10 Executive Administration       1002 Open World      10000000
     20 Information Technology         2001 DB11gR2           900000
```

You query a varray column in the same way that you query a nested table column.

In the first example in the slide, the collections are 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.

In the second example, the output is unnested 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.

Quiz

Which of the following collections is a set of key-value pairs, where each key is unique and is used to locate a corresponding value in the collection?

a. Associative arrays
b. Nested Table
c. Varray
d. Semsegs

Answer: a

Quiz

Which of the following collections can be stored in the database?

a. Associative arrays
b. Nested table
c. Varray

Answer: b, c

Quiz

Which of the following collections can be stored inline?

a. Associative arrays

b. Nested table

c. Varray

Answer: c

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

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 varrays in the database. Nested tables, varrays, and associative arrays can be used in a PL/SQL program.

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

Practice 4: Overview

This practice covers analyzing collections.

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

Use the `OE` schema for this practice.

5 Using Collections

Objectives

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

- Use collection methods
- Manipulate collections
- Distinguish between the different types of collections and when to use them
- Use PL/SQL bind types

In this lesson, you learn about using collections.

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, and then use the same types across many applications.

Lesson Agenda

- Working with collections
- Programming for collection exceptions
- Summarizing collections
- Using PL/SQL bind types

Working with Collections in PL/SQL

- You can declare collections as the 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
  PROCEDURE allocate_new_proj_list
    (p_dept_id NUMBER, p_name VARCHAR2, p_budget NUMBER);
  FUNCTION get_dept_project (p_dept_id NUMBER)
    RETURN typ_projectlist;
  PROCEDURE update_a_project
    (p_deptno NUMBER, p_new_project typ_Project,
     p_position NUMBER);
  FUNCTION manipulate_project (p_dept_id NUMBER)
    RETURN typ_projectlist;
  FUNCTION check_costs (p_project_list typ_projectlist)
    RETURN boolean;
END manage_dept_proj;
```

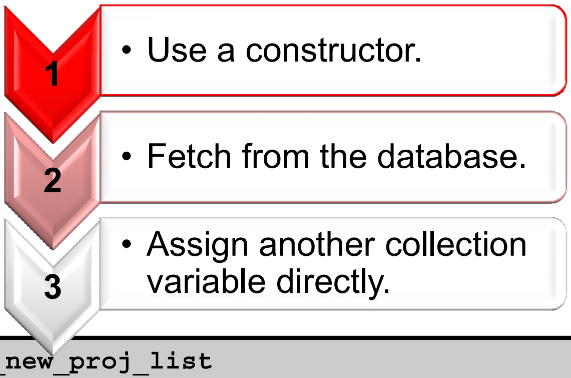
There are several points about collections that you must know when working with them:

- You can declare collections as the formal parameters of functions and procedures. Thus, 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.

This is the package body for the varray examples shown on the following pages.

```
CREATE OR REPLACE PACKAGE BODY manage_dept_proj
AS
  PROCEDURE allocate_new_proj_list
    (p_dept_id NUMBER, p_name VARCHAR2, p_budget NUMBER)
  IS
    v_accounting_project typ_projectlist;
  BEGIN   -- this example uses a constructor
    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
      (p_dept_id, p_name, p_budget, v_accounting_project);
  END allocate_new_proj_list;

  FUNCTION get_dept_project (p_dept_id NUMBER)
    RETURN typ_projectlist
  IS
    v_accounting_project typ_projectlist;
  BEGIN
   -- this example uses a fetch from the database
     SELECT  projects
       INTO  v_accounting_project
       FROM  department
       WHERE dept_id = p_dept_id;
     RETURN v_accounting_project;
  END get_dept_project;

PROCEDURE update_a_project
    (p_deptno NUMBER, p_new_project typ_Project,
     p_position NUMBER)
  IS
     v_my_projects typ_ProjectList;
  BEGIN
    v_my_projects := get_dept_project (p_deptno);
    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 update_a_project;
  FUNCTION manipulate_project (p_dept_id NUMBER)
    RETURN typ_projectlist
  IS
    v_accounting_project typ_projectlist;
    v_changed_list typ_projectlist;
  BEGIN
    SELECT  projects
       INTO  v_accounting_project
       FROM  department
       WHERE dept_id = p_dept_id;
  -- this example assigns one collection to another
    v_changed_list := v_accounting_project;
    RETURN v_changed_list;
  END manipulate_project;

  FUNCTION check_costs (p_project_list typ_projectlist)
    RETURN boolean
  IS
    c_max_allowed        NUMBER := 10000000;
    i                    INTEGER;
    v_flag               BOOLEAN := FALSE;
  BEGIN
    i := p_project_list.FIRST ;
    WHILE i IS NOT NULL LOOP
      IF p_project_list(i).cost > c_max_allowed then
        v_flag := TRUE;
        dbms_output.put_line (p_project_list(i).title ||
                         ' exceeded allowable budget.');
        RETURN TRUE;
      END IF;
    i := p_project_list.NEXT(i);
    END LOOP;
    RETURN null;
  END check_costs;

END manage_dept_proj;
```

Initializing Collections



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

```
PROCEDURE allocate_new_proj_list
   (p_dept_id NUMBER, p_name VARCHAR2, p_budget NUMBER)
 IS
   v_accounting_project typ_projectlist;
 BEGIN
 -- this example uses a constructor
   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(p_dept_id, p_name, p_budget, v_accounting_project);
 END allocate_new_proj_list;
```

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 methods:

- 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 in the slide, 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

1

```
FUNCTION get_dept_project (p_dept_id NUMBER)
   RETURN typ_projectlist
 IS
   v_accounting_project typ_projectlist;
 BEGIN  -- this example uses a fetch from the database
    SELECT  projects INTO  v_accounting_project
      FROM  department WHERE dept_id = p_dept_id;
    RETURN v_accounting_project;
 END get_dept_project;
```

2

```
FUNCTION manipulate_project (p_dept_id NUMBER)
   RETURN typ_projectlist
 IS
   v_accounting_project typ_projectlist;
   v_changed_list typ_projectlist;
 BEGIN
   SELECT  projects INTO  v_accounting_project
      FROM  department WHERE dept_id = p_dept_id;
 -- this example assigns one collection to another
   v_changed_list := v_accounting_project;
   RETURN v_changed_list;
 END manipulate_project;
```

ORACLE

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

In the second example, the entire content of one collection variable is assigned to another collection variable.

Referencing Collection Elements

```
-- sample caller program to the manipulate_project function
DECLARE
  v_result_list typ_projectlist;
BEGIN
  v_result_list := manage_dept_proj.manipulate_project(10);
  FOR i IN 1..v_result_list.COUNT LOOP
    dbms_output.put_line('Project #: '
                              ||v_result_list(i).project_no);
    dbms_output.put_line('Title: '||v_result_list(i).title);
    dbms_output.put_line('Cost: ' ||v_result_list(i).cost);
  END LOOP;

END;
```

```
Project #: 1001
Title: Travel Monitor
Cost: 400000
Project #: 1002
Title: Open World
Cost: 10000000
```

In the example in the slide, the code calls the `MANIPULATE PROJECT` function in the `MANAGE_DEPT_PROJ` package. Department 10 is passed in as the parameter. The output shows the varray element values for the `PROJECTS` column in the `DEPARTMENT` table for department 10.

Although the value of 10 is hard-coded, you can have a form interface to query the user for a department value that can then be passed into the routine.

Using Collection Methods

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

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

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

Commonly Used Collection Methods

Function or Procedure	Description
`EXISTS`	Returns `TRUE` if and only if specified element of varray or nested table exists
`COUNT`	Returns the number of elements that a collection contains
`LIMIT`	Returns the maximum number of elements that a collection can have
`FIRST and LAST`	Returns the first and last (smallest and largest) indexes in a collection
`PRIOR and NEXT`	`PRIOR(n)` returns the index that precedes index n in a collection; `NEXT(n)` returns the index that follows index n.
`EXTEND`	Appends one null element. `EXTEND(n)` appends n elements; `EXTEND(n, i)` appends n copies of the i th 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:** This does not work on varrays.

The slide lists some of the collection methods that you can use. You have already seen a few in the preceding examples.

Using Collection Methods

Traverse collections with the following methods:

```
FUNCTION check_costs (p_project_list typ_projectlist)
    RETURN boolean
  IS
    c_max_allowed        NUMBER := 10000000;
    i                    INTEGER;
    v_flag               BOOLEAN := FALSE;
  BEGIN
    i := p_project_list.FIRST ;
    WHILE i IS NOT NULL LOOP
      IF p_project_list(i).cost > c_max_allowed then
        v_flag := TRUE;
        dbms_output.put_line (p_project_list(i).title || '
                              exceeded allowable budget.');
        RETURN TRUE;
      END IF;
    i := p_project_list.NEXT(i);
    END LOOP;
    RETURN null;
  END check_costs;
```

In the example in the slide, the `FIRST` method finds the smallest index number, and the `NEXT` method traverses the collection starting at the first index.

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

```
-- sample caller program to check_costs
set serveroutput on
DECLARE
  v_project_list typ_projectlist;
BEGIN
  v_project_list := typ_ProjectList(
    typ_Project (1,'Dsgn New Expense Rpt', 3250),
    typ_Project (2, 'Outsource Payroll', 120000),
    typ_Project (3, 'Audit Accounts Payable',14250000));
  IF manage_dept_proj.check_costs(v_project_list) THEN
    dbms_output.put_line('Project rejected: overbudget');
  ELSE
    dbms_output.put_line('Project accepted, fill out forms.');
  END IF;
END;
```

```
Audit Accounts Payable exceeded allowable budget.
Project rejected: overbudget
```

V_PROJECT_LIST variable:

```
PROJECT_NO  TITLE                          COSTS
         1 Dsgn New Expense Rpt            3250
         2 Outsource Payroll             120000
         3 Audit Accounts Payable      14250000
```

The code shown in the slide calls the CHECK_COSTS function (shown on the previous page). The CHECK_COSTS function accepts a varray parameter and returns a Boolean value. If it returns true, the costs for a project element are too high. The maximum budget allowed for a project element is defined by the C_MAX_ALLOWED constant in the function.

A project with three elements is constructed and passed to the CHECK_COSTS function. The CHECK_COSTS function returns true, because the third element of the varray exceeds the value of the maximum allowed costs.

Although the sample caller program has the varray values hard-coded, you could have some sort of form interface where the user enters the values for projects and the form calls the CHECK_COSTS function.

Manipulating Individual Elements

```
PROCEDURE update_a_project
    (p_deptno NUMBER, p_new_project typ_Project, p_position NUMBER)
  IS
     v_my_projects typ_ProjectList;
  BEGIN
    v_my_projects := get_dept_project (p_deptno);
    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; -- insert new one
   UPDATE department SET projects = v_my_projects
     WHERE dept_id = p_deptno;
  END update_a_project;
```

You must use PL/SQL procedural statements to reference the individual elements of a varray in an `INSERT`, an `UPDATE`, or a `DELETE` statement. In the example shown in the slide, the `UPDATE_A_PROJECT` procedure inserts a new project into a department's project list at a given position, and then updates the `PROJECTS` column with the newly entered value that is placed within the old collection values.

This code essentially shuffles the elements of a project so that you can insert a new element in a particular position.

Manipulating Individual Elements

```
-- check the table prior to the update:
SELECT d2.dept_id, d2.name, d1.*
FROM department d2, TABLE(d2.projects) d1;
```

```
DEPT_ID NAME                     PROJECT_NO TITLE                        COST
------- ------------------------ ---------- ---------------------------- --
     10 Executive Administration       1001 Travel Monitor             400000
     10 Executive Administration       1002 Open World               10000000
     20 Information Technology         2001 DB11gR2                    900000
```

```
-- caller program to update_a_project
BEGIN
  manage_dept_proj.update_a_project(20,
    typ_Project(2002, 'AQM', 80000), 2);
END;
```

```
-- check the table after the update:
SELECT d2.dept_id, d2.name, d1.*
FROM department d2, TABLE(d2.projects) d1;
```

```
DEPT_ID NAME                     PROJECT_NO TITLE                        COST
------- ------------------------ ---------- ---------------------------- --
     10 Executive Administration       1001 Travel Monitor             400000
     10 Executive Administration       1002 Open World               10000000
     20 Information Technology         2001 DB11gR2                    900000
     20 Information Technology         2002 AQM                         80000
```

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.

The third code box shown in the slide identifies that a project element should be added to the second position for project 2002 in department 20.

If you execute the following code, the AQM project element is shuffled to position 3 and the CQN project element is inserted at position 2.

```
BEGIN
  manage_dept_proj.update_a_project(20,
    typ_Project(2003, 'CQN', 85000), 2);
END;
```

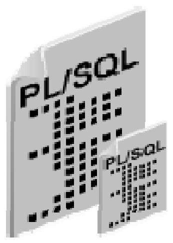
What happens if you request a project element to be inserted at position 5?

Querying a Collection Using the `TABLE` Operator

A collection can be queried if the following are true:

The data type of the collection was either created at the schema level or declared in a package specification.

In the query `FROM` clause, the collection appears in `table_collection_expression` as the argument of the `TABLE` operator.



A collection can be queried if the following are true:

- The data type of the collection was either created at the schema level or declared in a package specification.
- In the query `FROM` clause, the collection appears in `table_collection_expression` as the argument of the `TABLE` operator. The data type of the collection element is either a scalar data type that SQL supports or a record type in which every field has a data type that SQL supports.

Querying a Collection with Static SQL

```
CREATE OR REPLACE PACKAGE pkg AUTHID CURRENT_USER AS
  TYPE rec IS RECORD(f1 NUMBER, f2 VARCHAR2(30));
  TYPE mytab IS TABLE OF rec INDEX BY pls_integer;
END;

DECLARE
  v1 pkg.mytab;
  v2 pkg.rec;
  c1 SYS_REFCURSOR;
BEGIN
FOR i in 100..200 LOOP
   SELECT
   employee_id, last_name INTO v1(i)
   FROM employees WHERE employee_id = i;
END LOOP;
  OPEN c1 FOR SELECT * FROM TABLE(v1);
  FETCH c1 INTO v2;
  CLOSE c1;
END;
/
```

In the code example, the cursor variable `c1` is associated with a query of a nested table of records `(Select * from TABLE(v1))`. The nested table type, `mytab`, is declared in a package specification.

Querying a Collection with the `TABLE` Operator

```
CREATE OR REPLACE function GET_EMPS(P_JOB_ID  JOBS.JOB_ID%TYPE)
RETURN EMP_TYPE_LIST
IS
...
CURSOR C_EMP(C_JOB_ID  JOBS.JOB_ID%TYPE) IS
SELECT DEPARTMENT_ID,LAST_NAME,SALARY,J.JOB_ID,JOB_TITLE
FROM EMPLOYEES E, JOBS J
WHERE E.JOB_ID=J.JOB_ID AND J.JOB_ID=C_JOB_ID;
BEGIN
OPEN C_EMP(P_JOB_ID);
...
RETURN EMPS;
END;
/

SELECT * FROM TABLE(GET_EMPS('IT_PROG'))
/
SELECT  D.DEPARTMENT_NAME,E.*
FROM
TABLE(get_emps('IT_PROG')) E, DEPARTMENTS D
WHERE E.DEPARTMENT_ID=D.DEPARTMENT_ID
/
```

The code example in the slide shows the technique of querying a collection returned by a PL/SQL function by using the `TABLE` operator.

The complete example is illustrated on the following page.

Execute the following statements using your HR connection:

```
DROP TYPE EMP_TYPE_LIST ;
CREATE OR REPLACE TYPE EMP_TYPE AS OBJECT
(
 DEPARTMENT_ID            NUMBER(6),
 LAST_NAME               VARCHAR2(25),
 SALARY                  NUMBER(8,2),
 JOB_ID                  VARCHAR2(30),
 JOB_TITLE               VARCHAR2(30));
/

CREATE OR REPLACE TYPE EMP_TYPE_LIST AS TABLE OF EMP_TYPE;
/

CREATE OR REPLACE function
GET_EMPS(P_JOB_ID  JOBS.JOB_ID%TYPE)
RETURN EMP_TYPE_LIST
IS
EMPS  EMP_TYPE_LIST:=EMP_TYPE_LIST();
R EMP_TYPE:=EMP_TYPE(NULL,NULL,NULL,NULL,NULL);
i pls_integer:=0;
CURSOR C_EMP(C_JOB_ID  JOBS.JOB_ID%TYPE) IS
SELECT DEPARTMENT_ID,LAST_NAME,SALARY,J.JOB_ID,JOB_TITLE
FROM EMPLOYEES E, JOBS J
WHERE E.JOB_ID=J.JOB_ID AND J.JOB_ID=C_JOB_ID;
BEGIN
OPEN C_EMP(P_JOB_ID);
LOOP
FETCH C_EMP INTO
R.DEPARTMENT_ID,R.LAST_NAME,R.SALARY,R.JOB_ID,R.JOB_TITLE;
EXIT WHEN C_EMP%NOTFOUND;
i:=i+1;
emps.extend;
EMPS(I):=R;
END LOOP;
RETURN EMPS;
END;
/
```

Check the result:

```
SELECT * FROM TABLE(GET_EMPS('IT_PROG'));

SELECT  D.DEPARTMENT_NAME,E.*
FROM
TABLE(get_emps('IT_PROG')) E, DEPARTMENTS D
WHERE E.DEPARTMENT_ID=D.DEPARTMENT_ID;
```

Lesson Agenda

- Working with collections
- Programming for collection exceptions
- Summarizing collections
- Using PL/SQL bind types

Avoiding Collection Exceptions

Common exceptions with collections:

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

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.

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

Avoiding Collection Exceptions: Example

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
...
```

- 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 an element that was deleted.

Lesson Agenda

- Working with collections
- Programming for collection exceptions
- Summarizing collections
- Using PL/SQL bind types

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 inline (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

- Use associative arrays when you need:
  - To collect information of unknown volume
  - Flexible subscripts (negative, nonsequential, or string based)
  - To pass the collection to and from the database server (Use associative arrays with the bulk constructs.)
- Use nested tables when you need:
  - Persistence
  - 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 the index values
  - You need to delete or update some, not all, elements simultaneously
  - You would usually create a separate lookup table with multiple entries for each row of the main table and access it through join queries

Guidelines for 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.

After deleting the elements, release the unused memory with `DBMS_SESSION.FREE_UNUSED_USER_MEMORY`

- Because varray data is stored inline (in the same tablespace), retrieving and storing varrays involve 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. Thus, 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 the collection will not contain more than 10 elements in each row, you can use a varray with a limit of 10.
- 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.
- After deleting the elements, you can release the unused memory with the `DBMS_SESSION.FREE_UNUSED_USER_MEMORY` procedure.

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

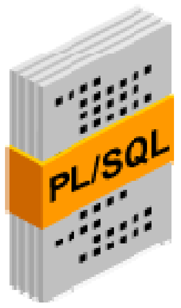
Lesson Agenda

- Working with collections
- Programming for collection exceptions
- Summarizing collections
- Using PL/SQL bind types

PL/SQL Bind Types

A PL/SQL anonymous block, a `SQL CALL` statement, or a SQL query can invoke a PL/SQL function that has parameters of the following types:

- Boolean
- Record declared in a package specification
- Collection declared in a package specification



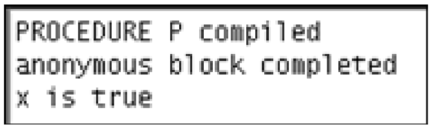
In Oracle Database 11*g*, when PL/SQL invoked SQL, a value could not be bound if its data type was Boolean or a collection or record type that was declared in a package specification. This restriction also held when the invoked SQL was a PL/SQL anonymous block. Therefore, PL/SQL could not dynamically invoke a PL/SQL subprogram that had a formal parameter whose data type was Boolean or a collection or record type declared in a package specification.

In Oracle Database Release 12*c* and later releases, these restrictions have been removed to enhance both the usability and performance aspects related to such PL/SQL usage.

Note: The `Index-by-VarChar2` table still cannot be used as a bind type, whereas Boolean, record declared in a package specification, and collection declared in a package specification can.

Subprogram with a `BOOLEAN` Parameter

```
CREATE OR REPLACE PROCEDURE p (x BOOLEAN) AUTHID
    CURRENT_USER AS
BEGIN
  IF x THEN
    DBMS_OUTPUT.PUT_LINE('x is true');
  END IF;
END;
/
DECLARE
  b  BOOLEAN := TRUE;
BEGIN
  p(b);
END;
/
```



```
PROCEDURE P compiled
anonymous block completed
x is true
```

In the code example shown in the slide, the dynamic PL/SQL block is an anonymous PL/SQL block that invokes a subprogram that has a formal parameter of the PL/SQL data type `BOOLEAN`.

Subprogram with a Record Parameter

```
CREATE OR REPLACE PACKAGE pkg AUTHID CURRENT_USER AS
   TYPE rec IS RECORD (n1 NUMBER, n2 NUMBER);
   PROCEDURE p_rectest (x rec, y NUMBER, z NUMBER);
END pkg;
/
CREATE OR REPLACE PACKAGE BODY pkg AS
   PROCEDURE p_rectest (x rec, y NUMBER, z NUMBER) AS
  BEGIN
    x.n1 := y;
    x.n2 := z;
  END p_rectest;
END pkg;
/
```

```
DECLARE
  r       pkg.rec;
  dyn_str VARCHAR2(3000);
BEGIN
  dyn_str := 'BEGIN p_rectest(:x, 6, 8); END;';
  EXECUTE IMMEDIATE dyn_str USING r;
  ....
); ....
```

For PL/SQL record binds, the bind metadata will contain Type Object Identifier (TOID) and Type Descriptor Object (TDO). A TOID will always be generated for all record types. The assignment to another variable of a record type will be done by comparing the TOID of source and the target variables. Any PL/SQL-only constraints, such as range constraints on `pls_integer`, will be ignored.

The code example shows that an anonymous dynamic PL/SQL block invokes the subprogram `p_rectest`, which has a formal parameter of the PL/SQL data type `RECORD`. The record type is declared in a package specification, and the subprogram is declared in the package specification and defined in the package body.

Subprogram with a Parameter of PL/SQL Collection Type

```
CREATE OR REPLACE PACKAGE pkg AUTHID CURRENT USER AS
   TYPE number_names IS TABLE OF VARCHAR2(5)
    INDEX BY PLS_INTEGER;
   PROCEDURE print_number_names (x number_names);
END pkg;
/
CREATE OR REPLACE PACKAGE BODY pkg AS
  PROCEDURE print_number_names (x number_names)  IS
  BEGIN
    FOR i IN 0..9 LOOP
      DBMS_OUTPUT.PUT_LINE(x(i));
    END LOOP;
  END;
END pkg;
```

```
DECLARE
  digit_names   pkg.number_names;
  dyn_stmt      VARCHAR2(3000);
BEGIN
  ...
  dyn_stmt := 'BEGIN print_number_names(:x); END;';
  EXECUTE IMMEDIATE dyn_stmt USING digit_names;
END;..
```

In Oracle Database 11*g*, PL/SQL could query a collection only if its data type was declared at the schema level and the collection element was not a record.

In Oracle Database Release 12*c* and later releases, a PL/SQL anonymous block, a `SQL CALL` statement, or a SQL query can invoke a PL/SQL function that has a parameter of PL/SQL collection type declared in a package specification.

In the code example, the dynamic PL/SQL block is an anonymous PL/SQL block that invokes a subprogram that has a formal parameter of the PL/SQL collection type associative array indexed by `PLS_INTEGER`. The collection type (number_names) is declared in a package specification, and the subprogram is declared in the package specification and defined in the package body.

Quiz

Which of the following collection method is used for traversing a collection?

a. `EXISTS`

b. `COUNT`

c. `LIMIT`

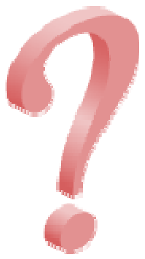
d. `FIRST`

Answer: d

Quiz

A PL/SQL anonymous block, a `SQL CALL` statement, or a SQL query can invoke a PL/SQL function that has parameters of the types:

a. Boolean
b. Record declared in the package specification
c. Collection declared in the package specification
d. All of the above



Answer: d

Quiz

In PL/SQL binds, Type Object Identifier (TOID) will always be generated for all record types.

a. True

b. False

Answer: a

Summary

In this lesson, you should have learned how to:

- Access collection elements
- Use collection methods in PL/SQL
- Identify raised exceptions with collections
- Decide which collection type is appropriate for each scenario

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 varrays 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 the collection elements, use predefined collection methods, and use the exceptions that are commonly encountered with collections.

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

Practice 5: Overview

This practice covers using collections.

In this practice, you write a PL/SQL package to manipulate the collection.
Use the `OE` schema for this practice.

6

Manipulating Large Objects

Objectives

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

- Create and maintain `LOB` data types
- Differentiate between internal and external `LOB`s
- Use the `DBMS_LOB` PL/SQL package
- Describe the use of temporary `LOB`s
- Describe SecureFile `LOB`

Databases have long been used to store large objects. However, the mechanisms built into databases have never been as useful as the large object (`LOB`) data types that have been provided since Oracle8. This lesson describes the characteristics of the new data types, comparing and contrasting them with the earlier data types. Examples, syntax, and issues regarding the `LOB` types are also presented.

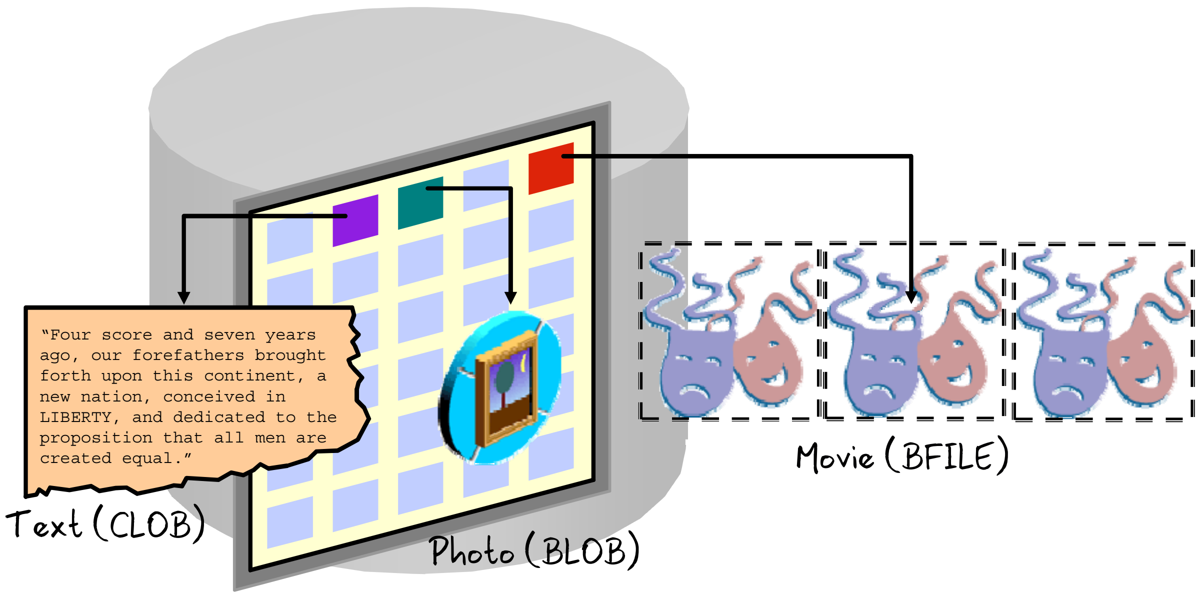
Note: A `LOB` is a data type and should not be confused with an object type.

Lesson Agenda

- Introduction to LOBs
- Managing BFILEs by using the DBMS_LOB package
- Manipulating LOB data
- Using temporary LOBs
- Using SecureFile LOB

What Is a LOB?

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


"Four score and seven years ago, our forefathers brought forth upon this continent, a new nation, conceived in LIBERTY, and dedicated to the proposition that all men are created equal."
Text (CLOB)
Photo (BLOB)
Movie (BFILE)

A LOB is a data type that is used to store large, unstructured data (such as text, graphic images, video clippings, and so on). Structured data, such as a customer record, can be a few hundred bytes large, but even small amounts of multimedia data can be thousands of times larger. Also, multimedia data may reside in operating system (OS) files, which needs to be accessed from a database.

There are four large object data types:

- BLOB represents a binary large object, such as a video clip.
- CLOB represents a character large object.
- NCLOB represents a multiple-byte character large object.
- BFILE represents a binary file stored in an OS binary file outside the database. The BFILE column or attribute stores a file locator that points to the external file.

LOBs are characterized in two ways: according to their interpretations by the Oracle server (binary or character) and their storage aspects. LOBs can be stored internally (inside the database) or in host files. There are two categories of LOBs:

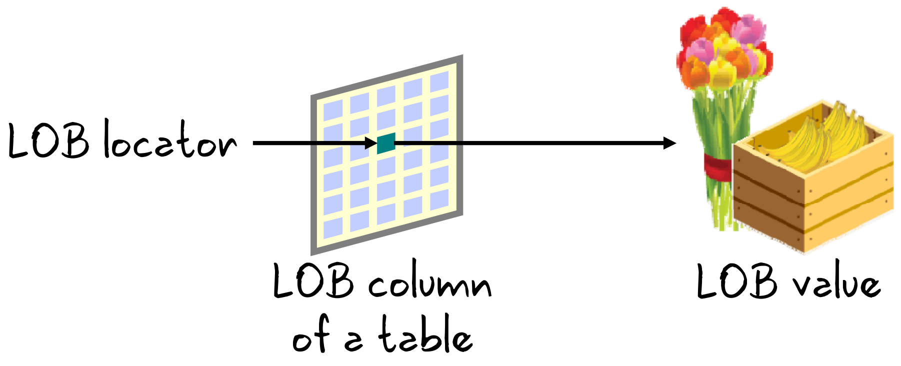
- **Internal LOBs (CLOB, NCLOB, BLOB):** Stored in the database
- **External files (BFILE):** Stored outside the database

Oracle Database performs implicit conversion between `CLOB` and `VARCHAR2` data types. The other implicit conversions between `LOB`s are not possible. For example, if the user creates a table `T` with a `CLOB` column and a table `S` with a `BLOB` column, the data is not directly transferable between these two columns.

`BFILE`s can be accessed only in read-only mode from an Oracle server.

Components of a LOB

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



There are two parts to a LOB:

- **LOB value:** The data that constitutes the real object being stored
- **LOB locator:** A pointer to the location of the LOB value that is stored in the database

Regardless of where the LOB value is stored, a locator is stored in the row. You can think of a LOB locator as a pointer to the actual location of the LOB value.

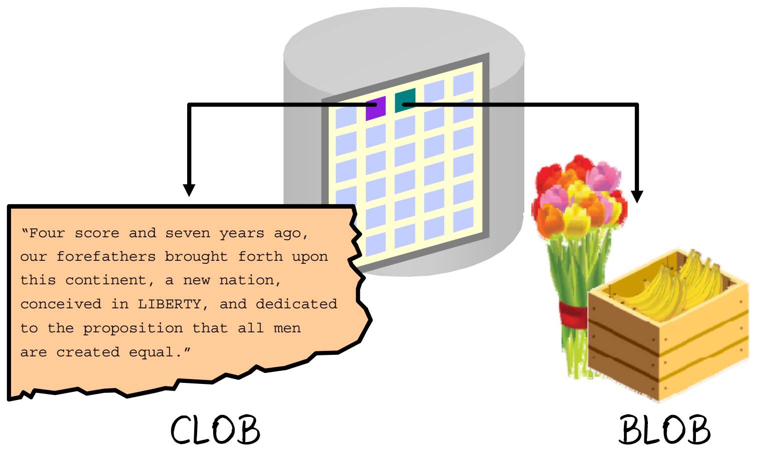
A LOB column does not contain the data; it contains the locator of the LOB value.

When a user creates an internal LOB, the value is stored in the LOB segment and a locator to the out-of-line LOB value is placed in the LOB column of the corresponding row in the table. External LOBs store the data outside the database, so only a locator to the LOB value is stored in the table.

To access and manipulate LOBs without SQL data manipulation language (DML), you must create a LOB locator. The programmatic interfaces operate on the LOB values by using these locators in a manner similar to OS file handles.

Internal LOBs

The LOB value is stored in the database.


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

An internal LOB is stored in the Oracle server. A BLOB, NCLOB, or CLOB can be one of the following:

- An attribute of a user-defined type
- A column in a table
- A bind or host variable
- A PL/SQL variable, parameter, or result

Internal LOBs can take advantage of Oracle features, such as:

- Concurrency mechanisms
- Redo logging and recovery mechanisms
- Transactions with COMMIT or ROLLBACK

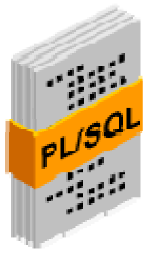
The BLOB data type is interpreted by the Oracle server as a bitstream, similar to the LONG RAW data type.

The CLOB data type is interpreted as a single-byte character stream.

The NCLOB data type is interpreted as a multiple-byte character stream, based on the byte length of the database national character set.

Managing Internal LOBs

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



To manage an internal LOB, perform the following steps:

1. Create and populate the table containing the LOB data type.
2. Declare and initialize the LOB locator in the program.
3. Use SELECT FOR UPDATE to lock the row containing the LOB into the LOB locator.
4. Manipulate the LOB with DBMS_LOB package procedures, OCI calls, Oracle Objects for OLE, Oracle precompilers, or JDBC by using the LOB locator as a reference to the LOB value. You can also manage LOBs through SQL.
5. Use the COMMIT command to make any changes permanent.

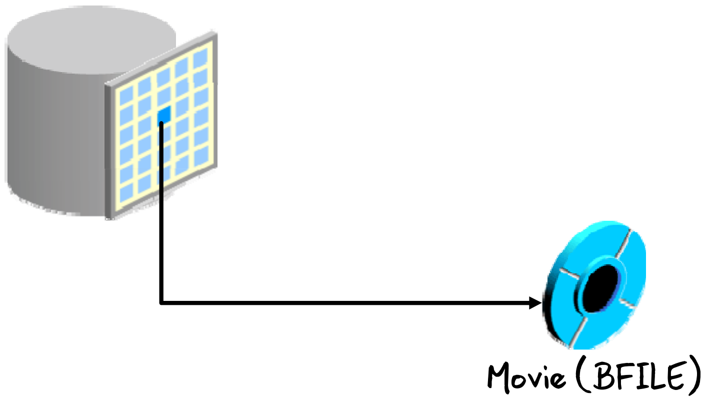
Lesson Agenda

- Introduction to `LOB`s
- Managing `BFILE`s by using the `DBMS_LOB` package
- Manipulating `LOB` data
- Using temporary `LOB`s
- Using `SecureFile LOB`

What Are BFILEs?

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

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

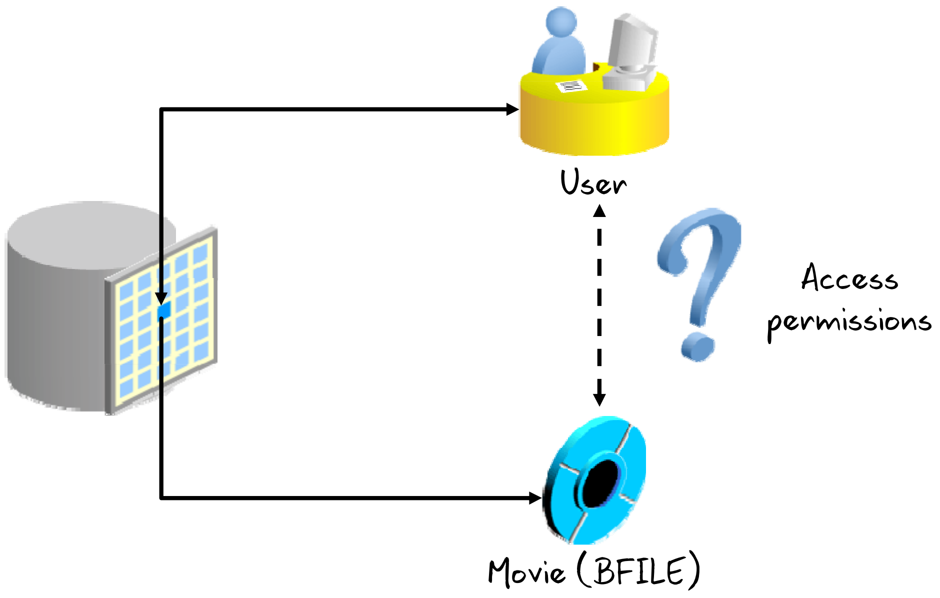


BFILEs are external large objects (LOBs) stored in OS files that are external to database tables. The BFILE data type stores a locator to the physical file. A BFILE can be in GIF, JPEG, MPEG, MPEG2, text, or other formats. The external LOBs may be located on hard disks, CD-ROMs, photo CDs, or other media, but a single LOB cannot extend from one medium or device to another. The BFILE data type is available so that database users can access the external file system. Oracle Database 12*c* provides:

- Definition of BFILE objects
- Association of BFILE objects with the corresponding external files
- Security for BFILEs

The remaining operations that are required for using BFILEs are possible through the DBMS_LOB package and OCI. BFILEs are read-only; they do not participate in transactions. Support for integrity and durability must be provided by the operating system. The file must be created and placed in the appropriate directory, giving the Oracle process privileges to read the file. When the LOB is deleted, the Oracle server does not delete the file. Administration of the files and the OS directory structures can be managed by the DBA, system administrator, or user. The maximum size of an external large object depends on the operating system, but cannot exceed 4 GB.

Securing `BFILE`s



Unauthenticated access to files on a server presents a security risk. Oracle Database can act as a security mechanism to shield the operating system from unsecured access while removing the need to manage additional user accounts on an enterprise computer system.

File Location and Access Privileges

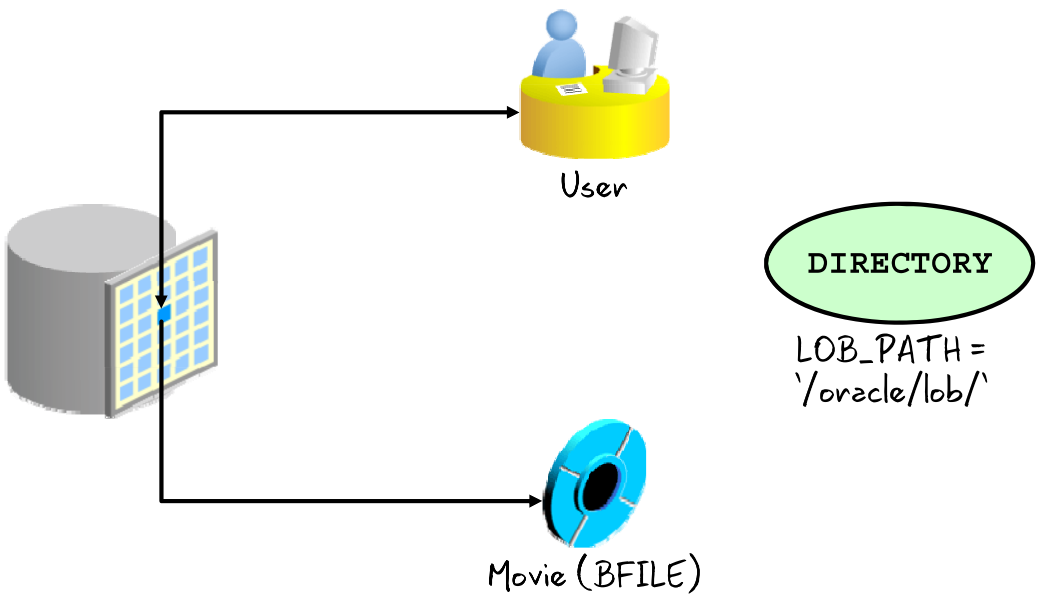
The file must reside on the machine where the database exists. A timeout to read a nonexistent `BFILE` is based on the OS value.

You can read a `BFILE` in the same way that you read an internal `LOB`. However, there could be restrictions related to the file itself, such as:

- Access permissions
- File system space limits
- Non-Oracle manipulations of files
- OS maximum file size

Oracle Database does not provide transactional support on `BFILE`s. Any support for integrity and durability must be provided by the underlying file system and the OS. Oracle backup and recovery methods support only the `LOB` locators, not the physical `BFILE`s.

What Is a DIRECTORY?



A DIRECTORY is a nonschema database object that enables the administration of access and usage of BFILEs in Oracle Database. To associate an OS file with a BFILE, you should first create a DIRECTORY object that is an alias for the full path name to the OS file.

By granting users suitable privileges for these items, you can provide secure access to files in the corresponding directories on a user-by-user basis.

The DIRECTORY object is created by the DBA (or a user with the CREATE ANY DIRECTORY privilege). The privileges may differ from those defined for the DIRECTORY object and could change after creation of the DIRECTORY object. Create DIRECTORY objects by using the following guidelines:

- Directories should point to paths that do not contain database files, because tampering with these files could corrupt the database.
- The CREATE ANY DIRECTORY and DROP ANY DIRECTORY system privileges should be used carefully and not granted to users indiscriminately.
- DIRECTORY objects are not schema objects; all are owned by SYS.
- Create the directory paths with appropriate permissions on the OS before creating the DIRECTORY object. Oracle does not create the OS path.

- If you migrate the database to a different OS, you may have to change the path value of the `DIRECTORY` object.

Information about the `DIRECTORY` object that you create by using the `CREATE DIRECTORY` command is stored in the `DBA_DIRECTORIES` and `ALL_DIRECTORIES` data dictionary views.

Using the DBMS_LOB Package

Working with LOBs often requires the use of the Oracle-supplied DBMS_LOB package.

LOB data can be retrieved directly using SQL.

In PL/SQL, you can define a VARCHAR2 for a CLOB and a RAW for a BLOB.

DBMS_LOB provides routines to access and manipulate internal and external LOBs.

To work with LOBs, you may have to use the DBMS_LOB package. The package does not support any concurrency control mechanism for BFILE operations. The user is responsible for locking the row containing the destination internal LOB before calling subprograms that involve writing to the LOB value. These DBMS_LOB routines do not implicitly lock the row containing the LOB. The two constants, LOBMAXSIZE and FILE_READONLY, that are defined in the package specification are also used in the procedures and functions of DBMS_LOB; for example, use them to achieve the maximum level of purity in SQL expressions.
The DBMS_LOB functions and procedures can be broadly classified into two types: mutators and observers:

- The mutators can modify LOB values: APPEND, COPY, ERASE, TRIM, WRITE, FILECLOSE, FILECLOSEALL, and FILEOPEN.
- The observers can read LOB values: COMPARE, FILEGETNAME, INSTR, GETLENGTH, READ, SUBSTR, FILEEXISTS, and FILEISOPEN.

DBMS_LOB provides routines to access and manipulate internal and external LOBs:

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

DBMS_LOB.READ and DBMS_LOB.WRITE

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

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

Call the READ procedure to read and return piecewise a specified AMOUNT of data from a given LOB, starting from OFFSET. An exception is raised when no more data remains to be read from the source LOB. The value returned in AMOUNT is less than the one specified if the end of the LOB is reached before the specified number of bytes or characters can be read. In the case of CLOBs, the character set of data in BUFFER is the same as that in the LOB.

PL/SQL allows a maximum length of 32,767 for RAW and VARCHAR2 parameters. Ensure that the allocated system resources are adequate to support buffer sizes for the given number of user sessions. Otherwise, the Oracle server raises the appropriate memory exceptions.

Note: BLOB and BFILE return RAW; the others return VARCHAR2.

DBMS_LOB.WRITE

Call the WRITE procedure to write piecewise a specified AMOUNT of data into a given LOB, from the user-specified BUFFER, starting from an absolute OFFSET from the beginning of the LOB value.

Make sure (especially with multiple-byte characters) that the amount in bytes corresponds to the amount of buffer data. WRITE has no means of checking whether they match, and it will write AMOUNT bytes of the buffer contents into the LOB.

Managing `BFILE`s: Role of a DBA

The DBA or the system administrator:

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

Managing `BFILE`s requires cooperation between the database administrator and the system administrator, and then between the developer and the user of the files.

The database or system administrator must perform the following privileged tasks:

1. Create the operating system (OS) directory (as an Oracle user), and set permissions so that the Oracle server can read the contents of the OS directory. Load files into the OS directory.
2. Create a database `DIRECTORY` object that references the OS directory.
3. Grant the `READ` privilege on the database `DIRECTORY` object to the database users that require access to it.

Managing `BFILE`s: Role of a Developer

The developer or the user:

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

The designer, application developer, or user must perform the following tasks:

1. Create a database table containing a column that is defined as the `BFILE` data type.
2. Insert rows into the table by using the `BFILENAME` function to populate the `BFILE` column, associating the `BFILE` field to an OS file in the named `DIRECTORY`.
3. Write PL/SQL subprograms that:
   a. Declare and initialize the `BFILE LOB` locator
   b. Select the row and column containing the `BFILE` into the `LOB` locator
   c. Read the `BFILE` with a `DBMS_LOB` function by using the locator file reference

Preparing to Use `BFILES`

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

```
mkdir /home/oracle/labs/DATA_FILES/MEDIA_FILES
```

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

```
CREATE OR REPLACE DIRECTORY data_files AS
'/home/oracle/labs/DATA_FILES/MEDIA_FILES';
```

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

```
GRANT READ ON DIRECTORY data_files TO OE;
```

To use a `BFILE` within an Oracle table, you must have a table with a column of the `BFILE` data type. For the Oracle server to access an external file, the server must know the physical location of the file in the OS directory structure.

The database `DIRECTORY` object provides the means to specify the location of the `BFILE`s. Use the `CREATE DIRECTORY` command to specify the pointer to the location where your `BFILE`s are stored. You must have the `CREATE ANY DIRECTORY` privilege.

Syntax definition: `CREATE DIRECTORY` *`dir_name`* `AS` *`os_path`*`;`

In this syntax, *`dir_name`* is the name of the directory database object, and *`os_path`* specifies the location of the `BFILE`s.

The slide examples show the commands to set up:

- The physical directory (for example, `/temp/data_files`) in the OS
- A named `DIRECTORY` object, called `data_files`, that points to the physical directory in the OS
- The `READ` access right on the directory to be granted to users in the database that provides the privilege to read the `BFILE`s from the directory

Note: The value of the `SESSION_MAX_OPEN_FILES` database initialization parameter, which is set to `10` by default, limits the number of `BFILE`s that can be opened in a session.

Populating `BFILE` Columns with SQL

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

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

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

```
ALTER TABLE customers ADD video BFILE;
```

  – Update the column using the `BFILENAME` function:

```
UPDATE customers
 SET video = BFILENAME('DATA_FILES', 'Winters.avi')
WHERE customer_id = 448;
```

The `BFILENAME` function is a built-in function that you use to initialize a `BFILE` column, by using the following two parameters:

- *`directory_alias`* for the name of the database `DIRECTORY` object that references the OS directory containing the files
- *`filename`* for the name of the `BFILE` to be read

The `BFILENAME` function creates a pointer (or `LOB` locator) to the external file stored in a physical directory, which is assigned a directory alias name that is used in the first parameter of the function. Populate the `BFILE` column by using the `BFILENAME` function in either of the following:

- The `VALUES` clause of an `INSERT` statement
- The `SET` clause of an `UPDATE` statement

An `UPDATE` operation can be used to change the pointer reference target of the `BFILE`. A `BFILE` column can also be initialized to a `NULL` value and updated later with the `BFILENAME` function, as shown in the slide.

After the `BFILE` columns are associated with a file, subsequent read operations on the `BFILE` can be performed by using the PL/SQL `DBMS_LOB` package and OCI. However, these files are read-only when accessed through `BFILE`s. Therefore, they cannot be updated or deleted through `BFILE`s.

Populating a `BFILE` Column with PL/SQL

```
CREATE OR REPLACE PROCEDURE set_video(
  dir_alias VARCHAR2, custid NUMBER) IS
  filename VARCHAR2(40);
  file_ptr BFILE;
  CURSOR cust_csr IS
    SELECT cust_first_name FROM customers
    WHERE customer_id = custid FOR UPDATE;
BEGIN
  FOR rec IN cust_csr LOOP
    filename := rec.cust_first_name || '.gif';
    file_ptr := BFILENAME(dir_alias, filename);
    DBMS_LOB.FILEOPEN(file_ptr);
    UPDATE customers SET video = file_ptr
      WHERE CURRENT OF cust_csr;
    DBMS_OUTPUT.PUT_LINE('FILE: ' || filename ||
     ' SIZE: ' || DBMS_LOB.GETLENGTH(file_ptr));
    DBMS_LOB.FILECLOSE(file_ptr);
  END LOOP;
END set_video;
```

The slide example shows a PL/SQL procedure called `set_video`, which accepts the name of the directory alias referencing the OS file system as a parameter, and a customer ID. The procedure performs the following tasks:

- Uses a cursor `FOR` loop to obtain each customer record
- Sets the `filename` by appending `.gif` to the customer's `first_name`
- Creates an in-memory `LOB` locator for the `BFILE` in the `file_ptr` variable
- Calls the `DBMS_LOB.FILEOPEN` procedure to verify whether the file exists, and to determine the size of the file by using the `DBMS_LOB.GETLENGTH` function
- Executes an `UPDATE` statement to write the `BFILE` locator value to the `video BFILE` column
- Displays the file size returned from the `DBMS_LOB.GETLENGTH` function
- Closes the file by using the `DBMS_LOB.FILECLOSE` procedure

Suppose that you execute the following call:

```
EXECUTE set_video('DATA_FILES', 844)
```

The sample result is:

```
FILE: Alice.gif SIZE: 2619802
```

Using DBMS_LOB Routines with BFILEs

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

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

```
CREATE OR REPLACE FUNCTION get_filesize(p_file_ptr IN
OUT BFILE)
RETURN NUMBER IS
  v_file_exists BOOLEAN;
  v_length NUMBER:= -1;
BEGIN
  v_file_exists := DBMS_LOB.FILEEXISTS(p_file_ptr) = 1;
  IF v_file_exists THEN
    DBMS_LOB.FILEOPEN(p_file_ptr);
    v_length := DBMS_LOB.GETLENGTH(p_file_ptr);
    DBMS_LOB.FILECLOSE(p_file_ptr);
  END IF;
  RETURN v_length;
END;
/
```

The set_video procedure on the previous page terminates with an exception if a file does not exist. To prevent the loop from prematurely terminating, you could create a function, such as get_filesize, to determine whether a given BFILE locator references a file that actually exists on the server's file system. The DBMS_LOB.FILEEXISTS function accepts the BFILE locator as a parameter and returns an INTEGER with:

- A value 0 if the physical file does not exist
- A value 1 if the physical file exists

If the BFILE parameter is invalid, one of the following three exceptions may be raised:

- NOEXIST_DIRECTORY if the directory does not exist
- NOPRIV_DIRECTORY if the database processes do not have privileges for the directory
- INVALID_DIRECTORY if the directory was invalidated after the file was opened

In the get_filesize function, the output of the DBMS_LOB.FILEEXISTS function is compared with the value 1 and the result of the condition sets the BOOLEAN variable file_exists. The DBMS_LOB.FILEOPEN call is performed only if the file exists, thereby preventing unwanted exceptions from occurring. The get_filesize function returns a value of -1 if a file does not exist; otherwise, it returns the size of the file in bytes. The caller can take appropriate action with this information.

Lesson Agenda

- Introduction to `LOB`s
- Managing `BFILE`s by using the `DBMS_LOB` package
- Manipulating `LOB` data
- Using temporary `LOB`s
- Using `SecureFile LOB`

Initializing LOB Columns Added to a Table

- Add the LOB columns to an existing table by using ALTER TABLE:

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

- Create a tablespace where you will put a new table with the LOB columns:

```
CREATE TABLESPACE lob_tbs1
 DATAFILE 'lob_tbs1.dbf' SIZE 800M REUSE
EXTENT MANAGEMENT LOCAL
UNIFORM SIZE 64M
SEGMENT SPACE MANAGEMENT AUTO;
```

The contents of a LOB column are stored in the LOB segment, whereas the column in the table contains only a reference to that specific storage area called the LOB locator. In PL/SQL, you can define a variable of the LOB type, which contains only the value of the LOB locator. You can initialize the LOB locators by using the following functions:

- EMPTY_CLOB() function to a LOB locator for a CLOB column
- EMPTY_BLOB() function to a LOB locator for a BLOB column

Note: These functions create the LOB locator value and not the LOB content. In general, you use the DBMS_LOB package subroutines to populate the content. The functions are available in Oracle SQL DML, and are not part of the DBMS_LOB package.

LOB columns are defined by using SQL data definition language (DDL). You can add LOB columns to an existing table by using the ALTER TABLE statement.

You can also add LOB columns to a new table. It is recommended that you create a tablespace first, and then create the new table in that tablespace.

Initializing LOB Columns Added to a Table

Initialize the column LOB locator value with the DEFAULT option or the DML statements by using:

- EMPTY_CLOB() function for a CLOB column
- EMPTY_BLOB() function for a BLOB column

```
CREATE TABLE customer_profiles (
   id  NUMBER,
   full_name    VARCHAR2(45),
   resume       CLOB DEFAULT EMPTY_CLOB(),
   picture      BLOB DEFAULT EMPTY_BLOB())
   LOB(picture) STORE AS BASICFILE
     (TABLESPACE lob_tbs1);
```

The example in the slide shows that you can use the EMPTY_CLOB() and EMPTY_BLOB() functions in the DEFAULT option in a CREATE TABLE statement. Thus, the LOB locator values are populated in their respective columns when a row is inserted into the table and the LOB columns were not specified in the INSERT statement.

The CUSTOMER_PROFILES table is created. The PICTURE column holds the LOB data in the BasicFile format, because the storage clause identifies the format.

Populating LOB Columns

- Insert a row into a table with LOB columns:

```
INSERT INTO customer_profiles
  (id, full_name, resume, picture)
 VALUES (164, 'Charlotte Kazan', EMPTY_CLOB(), NULL);
```

- Initialize a LOB by using the EMPTY_BLOB() function:

```
UPDATE customer_profiles
 SET resume = 'Date of Birth: 8 February 1951',
    picture = EMPTY_BLOB()
 WHERE id = 164;
```

- Update a CLOB column:

```
UPDATE customer_profiles
 SET resume = 'Date of Birth: 1 June 1956'
 WHERE id = 150;
```

You can insert a value directly into a LOB column by using host variables in SQL or PL/SQL, 3GL-embedded SQL, or OCI. You can use the special EMPTY_BLOB() and EMPTY_CLOB() functions in INSERT or UPDATE statements of SQL DML to initialize a NULL or non-NULL internal LOB to empty. To populate a LOB column, perform the following steps:

1. Initialize the LOB column to a non-NULL value—that is, set a LOB locator pointing to an empty or populated LOB value. This is done by using the EMPTY_BLOB() and EMPTY_CLOB() functions.
2. Populate the LOB contents by using the DBMS_LOB package routines.

However, as shown in the slide examples, the two UPDATE statements initialize the resume LOB locator value and populate its contents by supplying a literal value. This can also be done in an INSERT statement. A LOB column can be updated to:

- Another LOB value
- A NULL value
- A LOB locator with empty contents by using the EMPTY_*LOB() built-in function

You can update the LOB by using a bind variable in embedded SQL. When assigning one LOB to another, a new copy of the LOB value is created. Use a SELECT FOR UPDATE statement to lock the row containing the LOB column before updating a piece of the LOB contents.

Writing Data to a LOB

- Create the procedure to read the MS Word files and load them into the LOB column.
- Call this procedure from the WRITE_LOB procedure (shown in the next slide).

```
CREATE OR REPLACE PROCEDURE loadLOBFromBFILE_proc
   (p_dest_loc IN OUT BLOB, p_file_name IN VARCHAR2,
    p_file_dir IN VARCHAR2)
IS
  v_src_loc  BFILE := BFILENAME(p_file_dir, p_file_name);
  v_amount   INTEGER := 4000;
BEGIN
  DBMS_LOB.OPEN(v_src_loc, DBMS_LOB.LOB_READONLY);
  v_amount := DBMS_LOB.GETLENGTH(v_src_loc);
  DBMS_LOB.LOADFROMFILE(p_dest_loc, v_src_loc, v_amount);
  DBMS_LOB.CLOSE(v_src_loc);
END loadLOBFromBFILE_proc;
```

The procedure shown in the slide is used to load data into the LOB column.

Before running the LOADLOBFROMBFILE_PROC procedure, you must set a directory object that identifies where the LOB files are stored externally. In this example, the Microsoft Word documents are stored in the DATA_FILES directory that was created earlier in this lesson.

The LOADLOBFROMBFILE_PROC procedure is used to read the LOB data into the PICTURE column in the CUSTOMER_PROFILES table.

In this example:

- DBMS_LOB.OPEN is used to open an external LOB in read-only mode.
- DBMS_LOB.GETLENGTH is used to find the length of the LOB value.
- DBMS_LOB.LOADFROMFILE is used to load the BFILE data into an internal LOB.
- DBMS_LOB.CLOSE is used to close the external LOB.

Note: The LOADLOBFROMBFILE_PROC procedure shown in the slide can be used to read both the SecureFile and BasicFile formats. SecureFile LOBs are discussed later in this lesson.

Writing Data to a LOB

Create the procedure to insert LOBs into the table:

```
CREATE OR REPLACE PROCEDURE write_lob
  (p_file IN VARCHAR2, p_dir IN VARCHAR2)
IS
 i    NUMBER;          v_fn VARCHAR2(15);
 v_ln VARCHAR2(40);    v_b  BLOB;
BEGIN
  DBMS_OUTPUT.ENABLE;
  DBMS_OUTPUT.PUT_LINE('Begin inserting rows...');
  FOR i IN 1 .. 30 LOOP
    v_fn:=SUBSTR(p_file,1,INSTR(p_file,'.')-1);
    v_ln:=SUBSTR(p_file,INSTR(p_file,'.')+1,LENGTH(p_file)-
               INSTR(p_file,'.')-4);
    INSERT INTO customer_profiles
      VALUES (i, v_fn, v_ln, EMPTY_BLOB())
      RETURNING picture INTO v_b;
    loadLOBFromBFILE_proc(v_b,p_file, p_dir);
    DBMS_OUTPUT.PUT_LINE('Row '|| i ||' inserted.');
  END LOOP;
  COMMIT;
END write_lob;
```

Before you write data to the LOB column, you must make the LOB column non-NULL. The LOB column must contain a locator that points to an empty or a populated LOB value. You can initialize a BLOB column value by using the EMPTY_BLOB() function as a default predicate.

The code shown in the slide uses the INSERT statement to initialize the locator. The LOADLOBFROMBFILE routine is then called and the LOB column value is inserted.

The write and read performance statistics for LOB storage is captured through output messages.

Writing Data to a LOB

```
CREATE OR REPLACE DIRECTORY resume_files AS
'/home/oracle/labs/DATA_FILES/RESUMES';
```

```
set serveroutput on
set verify on
set term on

timing start load_data
execute write_lob('karl.brimmer.doc',  'RESUME_FILES')
execute write_lob('monica.petera.doc', 'RESUME_FILES')
execute write_lob('david.sloan.doc',   'RESUME_FILES')
timing stop
```

1. The Microsoft Word files are stored in the `/home/oracle/labs/DATA_FILES/RESUMES` directory.
2. To read them into the `PICTURE` column in the `CUSTOMER_PROFILES` table, the `WRITE_LOB` procedure is called and the name of the `.doc` files is passed as a parameter.

Note: This script is run in SQL*Plus, because `TIMING` is a SQL*Plus option and is not available in SQL Developer.

The output is similar to the following:

```
timing start load_data
execute write_lob('karl.brimmer.doc', 'RESUME_FILES');
Begin inserting rows...
Row 1 inserted.
...
PL/SQL procedure successfully completed.

execute write_lob('monica.petera.doc', 'RESUME_FILES');
Begin inserting rows...
Row 1 inserted.
...
PL/SQL procedure successfully completed.

execute write_lob('david.sloan.doc', 'RESUME_FILES');
Begin inserting rows...
Row 1 inserted.
...
PL/SQL procedure successfully completed.

timing stop
timing for: load_data
Elapsed: 00:00:00.96
```

Reading LOBs from the Table

```
CREATE OR REPLACE  PROCEDURE lob_txt(file_name VARCHAR2,p_dir VARCHAR2 DEFAULT 'PLSQL_DIR')
IS
c CLOB:=null;
byte_count pls_integer;
fil BFILE:=BFILENAME(p_DIR,file_name);
v_dest_offset  integer:=1;
v_src_offset  integer:=1;
v_lang_context integer:=0;
v_warning integer;
BEGIN
c:=TO_CLOB(' ');
IF DBMS_LOB.FILEEXISTS(FIL)=1 then
  DBMS_LOB.FILEOPEN(fil,DBMS_LOB.FILE_READONLY);
   byte_count:=DBMS_LOB.GETLENGTH(fil);
  DBMS_OUTPUT.PUT_LINE('The length of the file:'||byte_count);
  DBMS_LOB.LOADCLOBFROMFILE
  (dest_lob => c,src_bfile => fil,amount => byte_count,dest_offset => v_dest_offset
  ,src_offset => v_src_offset,bfile_csid => 0,lang_context => v_lang_context
  ,warning => v_lang_context);
  DBMS_LOB.FILECLOSEALL;
  INSERT INTO lob_text VALUES (lob_seq.nextval,c);
 COMMIT;
ELSE
  DBMS_OUTPUT.PUT_LINE('The file does not exist ');
END IF;
END;
/
```

The procedure given in the slide uploads a text file into a CLOB type column in a table. To retrieve the records that were inserted, you can call the `LOB_TXT` procedure:

```
SET LONG 10000
set serveroutput on
exec lob_txt('java.sql')
select KEY,txt from lob_text;
```

Updating `LOB` by Using `DBMS_LOB` in PL/SQL

```
DECLARE
  v_lobloc CLOB;     -- serves as the LOB locator
  v_text   VARCHAR2(50) := 'Resigned = 5 June 2000';
  v_amount NUMBER ; -- amount to be written
  v_offset INTEGER; -- where to start writing
BEGIN
  SELECT resume INTO v_lobloc FROM customer_profiles
  WHERE id = 164 FOR UPDATE;
  v_offset := DBMS_LOB.GETLENGTH(v_lobloc) + 2;
  v_amount := length(v_text);
  DBMS_LOB.WRITE (v_lobloc, v_amount, v_offset, v_text);
  v_text := ' Resigned = 30 September 2000';
  SELECT resume INTO v_lobloc FROM customer_profiles
  WHERE id = 150 FOR UPDATE;
  v_amount := length(v_text);
  DBMS_LOB.WRITEAPPEND(v_lobloc, v_amount, v_text);
  COMMIT;
END;
```

In the example in the slide, the `LOBLOC` variable serves as the `LOB` locator, and the `AMOUNT` variable is set to the length of the text that you want to add. The `SELECT FOR UPDATE` statement locks the row and returns the `LOB` locator for the `RESUME LOB` column. Finally, the PL/SQL `WRITE` package procedure is called to write the text into the `LOB` value at the specified offset. `WRITEAPPEND` appends to the existing `LOB` value.

The example shows how to fetch a `CLOB` column in releases before Oracle9*i*. In those releases, it was not possible to fetch a `CLOB` column directly into a character column. The column value must be bound to a `LOB` locator, which is accessed by the `DBMS_LOB` package. An example later in this lesson shows that you can directly fetch a `CLOB` column by binding it to a character variable.

Checking the Space Usage of a LOB Table

```
CREATE OR REPLACE PROCEDURE check_space
IS
  l_fs1_bytes NUMBER;
  l_fs2_bytes NUMBER; ...
BEGIN
  DBMS_SPACE.SPACE_USAGE(
    segment_owner      => 'OE',
    segment_name       => 'CUSTOMER_PROFILES',
    segment_type       => 'TABLE',
    fs1_bytes          => l_fs1_bytes,
    fs1_blocks         => l_fs1_blocks,
    fs2_bytes          => l_fs2_bytes,
    fs2_blocks         => l_fs2_blocks, ...
  );
  DBMS_OUTPUT.ENABLE;
  DBMS_OUTPUT.PUT_LINE(' FS1 Blocks = '||l_fs1_blocks||'
     Bytes = '||l_fs1_bytes);
  DBMS_OUTPUT.PUT_LINE(' FS2 Blocks = '||l_fs2_blocks||'
     Bytes = '||l_fs2_bytes); …
  DBMS_OUTPUT.PUT_LINE('=======================================');
  DBMS_OUTPUT.PUT_LINE('Total Blocks =
     '||to_char(l_fs1_blocks + l_fs2_blocks …));
END;
/
```

To check the space usage in the disk blocks allocated to the LOB segment in the CUSTOMER_PROFILES table, use the CHECK_SPACE, as shown in the slide. This procedure calls the DBMS_SPACE package.

To execute the procedure, run the following command:

```
EXECUTE check_space
```

The output is as follows:

```
FS1 Blocks = 1     Bytes = 8192
FS2 Blocks = 0     Bytes = 0
FS3 Blocks = 1     Bytes = 8192
FS4 Blocks = 3     Bytes = 24576
Full Blocks = 0    Bytes = 0
=========================
Total Blocks =     5 ||
Total Bytes = 40960
PL/SQL procedure successfully completed.
```

Complete Code of the `CHECK_SPACE` Procedure

```
CREATE OR REPLACE PROCEDURE check_space
IS
  l_fs1_bytes NUMBER;   l_fs2_bytes NUMBER;
  l_fs3_bytes NUMBER;   l_fs4_bytes NUMBER;
  l_fs1_blocks NUMBER;  l_fs2_blocks NUMBER;
  l_fs3_blocks NUMBER;  l_fs4_blocks NUMBER;
  l_full_bytes NUMBER;  l_full_blocks NUMBER;
  l_unformatted_bytes NUMBER;
  l_unformatted_blocks NUMBER;
BEGIN
  DBMS_SPACE.SPACE_USAGE(
    segment_owner      => 'OE',
    segment_name       => 'CUSTOMER_PROFILES',
    segment_type       => 'TABLE',
    fs1_bytes          => l_fs1_bytes,
    fs1_blocks         => l_fs1_blocks,
    fs2_bytes          => l_fs2_bytes,
    fs2_blocks         => l_fs2_blocks,
    fs3_bytes          => l_fs3_bytes,
    fs3_blocks         => l_fs3_blocks,
    fs4_bytes          => l_fs4_bytes,
    fs4_blocks         => l_fs4_blocks,
    full_bytes         => l_full_bytes,
    full_blocks        => l_full_blocks,
    unformatted_blocks => l_unformatted_blocks,
    unformatted_bytes  => l_unformatted_bytes
   );
DBMS_OUTPUT.ENABLE;
  DBMS_OUTPUT.PUT_LINE(' FS1 Blocks = '||l_fs1_blocks||'
     Bytes = '||l_fs1_bytes);
  DBMS_OUTPUT.PUT_LINE(' FS2 Blocks = '||l_fs2_blocks||'
     Bytes = '||l_fs2_bytes);
  DBMS_OUTPUT.PUT_LINE(' FS3 Blocks = '||l_fs3_blocks||'
     Bytes = '||l_fs3_bytes);
  DBMS_OUTPUT.PUT_LINE(' FS4 Blocks = '||l_fs4_blocks||'
     Bytes = '||l_fs4_bytes);
  DBMS_OUTPUT.PUT_LINE('Full Blocks = '||l_full_blocks||'
     Bytes = '||l_full_bytes);
  DBMS_OUTPUT.PUT_LINE('=======================================
      =========');
  DBMS_OUTPUT.PUT_LINE('Total Blocks =
     '||to_char(l_fs1_blocks + l_fs2_blocks +
     l_fs3_blocks + l_fs4_blocks + l_full_blocks)||  ' ||
     Total Bytes = '|| to_char(l_fs1_bytes + l_fs2_bytes
     + l_fs3_bytes + l_fs4_bytes + l_full_bytes));
END;
```

Selecting `CLOB` Values by Using SQL

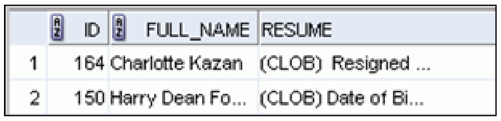
- Query:

```
SELECT id, full_name , resume -- CLOB
FROM customer_profiles
WHERE id IN (164, 150);
```

- Output in SQL*Plus:

```
 ID   FULL_NAME         RESUME
----- ---------------   ------------------
164   Charlotte Kazan   Date of Birth: 8 February 1951
                        Resigned = 5 June 2000
150   Harry Dean Fonda  Date of Birth: 1 June 1
                        956 Resigned = 30 September 2000
```

- Output in SQL Developer:



	ID	FULL_NAME	RESUME
1	164	Charlotte Kazan	(CLOB) Resigned ...
2	150	Harry Dean Fo...	(CLOB) Date of Bi...

It is possible to see the data in a `CLOB` column by using a `SELECT` statement. It is not possible to see the data in a `BLOB` or `BFILE` column by using a `SELECT` statement in SQL*Plus. You must use a tool that can display the binary information for a `BLOB`, as well as the relevant software for a `BFILE`—for example, you can use Oracle Forms.

Selecting `CLOB` Values by Using `DBMS_LOB`

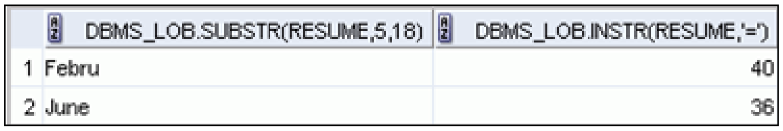
- `DBMS_LOB.SUBSTR (lob, amount, start_pos)`
- `DBMS_LOB.INSTR (lob, pattern)`

```
SELECT DBMS_LOB.SUBSTR (resume, 5, 18),
       DBMS_LOB.INSTR (resume,' = ')
FROM   customer_profiles
WHERE  id IN (150, 164);
```

- SQL*Plus

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

- SQL Developer



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

`DBMS_LOB.SUBSTR`

Use `DBMS_LOB.SUBSTR` to display part of a `LOB`. It is similar in functionality to the `SUBSTR` SQL function.

`DBMS_LOB.INSTR`

Use `DBMS_LOB.INSTR` to search for information within the `LOB`. This function returns the numerical position of the information.

Selecting CLOB Values in PL/SQL

```
SET LINESIZE 50 SERVEROUTPUT ON FORMAT WORD_WRAP

DECLARE
  text VARCHAR2(4001);
BEGIN
 SELECT resume INTO text
 FROM customer_profiles
 WHERE id = 150;
 DBMS_OUTPUT.PUT_LINE('text is: '|| text);
END;
/
```

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

The slide shows the code for accessing `CLOB` values that can be implicitly converted to `VARCHAR2`. When selected, the `RESUME` column value is implicitly converted from a `CLOB` to a `VARCHAR2` to be stored in the `TEXT` variable.

Removing LOBs

- Delete a row containing LOBs:

```
DELETE
FROM   customer_profiles
WHERE id = 164;
```

- Disassociate a LOB value from a row:

```
UPDATE customer_profiles
SET resume = EMPTY_CLOB()
WHERE id = 150;
```

A LOB instance can be deleted (destroyed) by using the appropriate SQL DML statements. The DELETE SQL statement deletes a row and its associated internal LOB value. To preserve the row and destroy only the reference to the LOB, you must update the row by replacing the LOB column value with NULL or an empty string, or by using the EMPTY_B/CLOB() function.

Note: Replacing a column value with NULL and using EMPTY_B/CLOB are not the same. Using NULL sets the value to null; using EMPTY_B/CLOB ensures that nothing is in the column.

A LOB is destroyed when the row containing the LOB column is deleted, when the table is dropped or truncated, or when all LOB data is updated.

You must explicitly remove the file associated with a BFILE by using the OS commands.

To erase part of an internal LOB, you can use DBMS_LOB.ERASE.

Quiz

The `BFILE` data type stores a locator to the physical file.

a. True

b. False

Answer: a

Quiz

Use the `BFILENAME` function to:

a. Create a `BFILE` column
b. Initialize a `BFILE` column
c. Update a `BFILE` column

Answer: b, c

Quiz

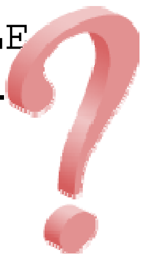
Which of the following statements are true?

a. You should initialize the LOB column to a non-`NULL` value by using the `EMPTY_BLOB()` and `EMPTY_CLOB()` functions.

b. Populate the LOB contents by using the `DBMS_LOB` package routines.

c. It is possible to see the data in a `CLOB` column by using a `SELECT` statement.

d. It is not possible to see the data in a `BLOB` or `BFILE` column by using a `SELECT` statement in SQL*Plus.



Answer: a, b, c, d

Lesson Agenda

- Introduction to `LOB`s
- Managing `BFILE`s by using the `DBMS_LOB` package
- Manipulating `LOB` data
- Using temporary `LOB`s
- Using `SecureFile LOB`

Temporary LOBs

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

Temporary LOBs provide an interface to support the creation and deletion of LOBs that act like local variables. Temporary LOBs can be BLOBs, CLOBs, or NCLOBs.

The following are the features of temporary LOBs:

- Data is stored in your temporary tablespace, not in tables.
- Temporary LOBs are faster than persistent LOBs, because they do not generate redo or rollback information.
- Temporary LOBs lookup is localized to each user's own session. Only the user who creates a temporary LOB can access it, and all temporary LOBs are deleted at the end of the session in which they were created.
- You can create a temporary LOB by using DBMS_LOB.CREATETEMPORARY.

Temporary LOBs are useful when you want to perform a transformational operation on a LOB (for example, changing an image type from GIF to JPEG). A temporary LOB is empty when created and does not support the EMPTY_B/CLOB functions.

Use the DBMS_LOB package to use and manipulate temporary LOBs.

Creating a Temporary LOB

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

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

The example in the slide shows a user-defined PL/SQL procedure, `is_templob_open`, which creates a temporary `LOB`. This procedure accepts a `LOB` locator as input, creates a temporary `LOB`, opens it, and tests whether the `LOB` is open.

The `is_templob_open` procedure uses the procedures and functions from the `DBMS_LOB` package as follows:

- The `CREATETEMPORARY` procedure is used to create the temporary `LOB`.
- The `ISOPEN` function is used to test whether a `LOB` is open: This function returns the value `1` if the `LOB` is open.
- The `FREETEMPORARY` procedure is used to free the temporary `LOB`. Memory increases incrementally as the number of temporary `LOB`s grows, and you can reuse the temporary `LOB` space in your session by explicitly freeing temporary `LOB`s.

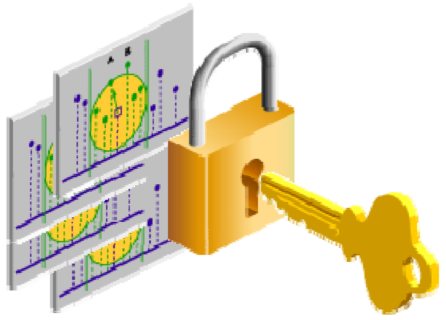
Lesson Agenda

- Introduction to `LOB`s
- Managing `BFILE`s by using the `DBMS_LOB` package
- Migrating `LONG` data types to `LOB`s
- Manipulating `LOB` data
- Using temporary `LOB`s
- Using `SecureFile` `LOB`

SecureFile LOBs

Oracle Database offers a reengineered large object (LOB) data type that:

- Improves performance
- Eases manageability
- Simplifies application development
- Offers advanced, next-generation functionality such as intelligent compression and transparent encryption



With SecureFile LOBs, the LOB data type is completely reengineered with dramatically improved performance, manageability, and ease of application development. This implementation also offers advanced, next-generation functionality such as intelligent compression and transparent encryption. This feature significantly strengthens the native content management capabilities of Oracle Database.

SecureFile LOBs were introduced to supplement the implementation of original BasicFile LOBs that are identified by the BASICFILE SQL parameter.

Storage of SecureFile LOBs

An efficient new storage paradigm is available in Oracle Database for LOB storage.

- If the SECUREFILE storage keyword appears in the CREATE TABLE statement, the new storage is used.
- If the BASICFILE storage keyword appears in the CREATE TABLE statement, the old storage paradigm is used.
- By default, the storage is BASICFILE, unless you modify the setting for the DB_SECUREFILE parameter in the init.ora file.

Starting with Oracle Database 11*g*, you have the option of using the new SecureFile storage paradigm for LOBs. You can specify the use of the new paradigm by using the SECUREFILE keyword in the CREATE TABLE statement. If that keyword is left out, the old storage paradigm for basic file LOBs is used. This is the default behavior.

You can modify the init.ora file and change the default behavior for the storage of LOBs by setting the DB_SECUREFILE initialization parameter. The following values are allowed:

- ALWAYS: Attempts to create all LOB files as SECUREFILES but creates any LOBs not in ASSM tablespaces as BASICFILE LOBs
- FORCE: Creates all LOBs in the system as SECUREFILE LOBs
- PERMITTED: The default; allows SECUREFILES to be created when specified with the SECUREFILE keyword in the CREATE TABLE statement
- NEVER: Creates LOBs that are specified as SECUREFILE LOBs as BASICFILE LOBs
- IGNORE: Ignores the SECUREFILE keyword and all SECUREFILE options

Creating a SecureFile `LOB`

- Create a tablespace for the `LOB` data:

```
-- have your dba do this:
CREATE TABLESPACE sf_tbs1
 DATAFILE 'sf_tbs1.dbf' SIZE 1500M REUSE
 AUTOEXTEND ON NEXT 200M
 MAXSIZE 3000M
 SEGMENT SPACE MANAGEMENT AUTO;
```

1

- Create a table to hold the `LOB` data:

```
CONNECT oe/oe
CREATE TABLE customer_profiles_sf
(id NUMBER,
 first_name VARCHAR2 (40),
 last_name  VARCHAR2 (80),
 profile_info  BLOB)
 LOB(profile_info) STORE AS SECUREFILE
 (TABLESPACE sf_tbs1);
```

2

To create a column to hold a `LOB` that is a SecureFile, you:

- Create a tablespace to hold the data
- Define a table that contains a `LOB` column data type that is used to store the data in the SecureFile format

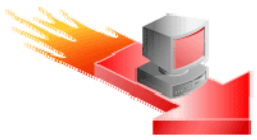
In the example shown in the slide:

1. The `sf_tbs1` tablespace is defined. This tablespace stores the `LOB` data in the SecureFile format. When you define a column to hold SecureFile data, you must have Automatic Segment Space Management (ASSM) enabled for the tablespace to support SecureFiles.
2. The `CUSTOMER_PROFILES_SF` table is created. The `PROFILE_INFO` column holds the `LOB` data in the SecureFile format, because the storage clause identifies the format.

Comparing Performance

Compare the performance on loading and reading `LOB` columns in the SecureFile and BasicFile formats:

Performance Comparison	Loading Data	Reading Data
SecureFile format	00:00:00.96	00:00:01.09
BasicFile format	00:00:01.68	00:00:01.15



In the examples shown in this lesson, the performance on loading and reading data in the `LOB` column of the SecureFile format `LOB` is faster than that of the BasicFile format `LOB`.

Quiz

Which of the following statements are true about temporary `LOB`s?

a. Data is stored in your temporary tablespace, not in tables.

b. Temporary `LOB`s are faster than persistent `LOB`s, because they do not generate redo or rollback information.

c. Only the user who creates a temporary `LOB` can access it, and all temporary `LOB`s are deleted at the end of the session in which they were created.

d. You can create a temporary `LOB` by using `DBMS_LOB.CREATETEMPORARY`.

Answer: a, b, c, d

Summary

In this lesson, you should have learned how to:

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

There are four `LOB` data types:

- A `BLOB` is a binary large object.
- A `CLOB` is a character large object.
- An `NCLOB` stores multiple-byte national character set data.
- A `BFILE` is a large object stored in a binary file outside the database.

`LOB`s can be stored internally (in the database) or externally (in an OS file). You can manage `LOB`s by using the `DBMS_LOB` package and its procedure.

Temporary `LOB`s provide an interface to support the creation and deletion of `LOB`s that act like local variables.

You learned about SecureFile format, and also learned that the performance of SecureFile format `LOB`s is faster than the BasicFile format `LOB`s.

Practice 6: Overview

This practice covers the following topics:

- Creating object types of the `CLOB` and `BLOB` data types
- Creating a table with the `LOB` data types as columns
- Using the `DBMS_LOB` package to populate and interact with the `LOB` data
- Setting up the environment for `LOB`s

In this practice, you create a table with both the `BLOB` and `CLOB` columns. Then, you use the `DBMS_LOB` package to populate the table and manipulate the data.

Use the `OE` schema for this practice.

7 Using Advanced Interface Methods

Objectives

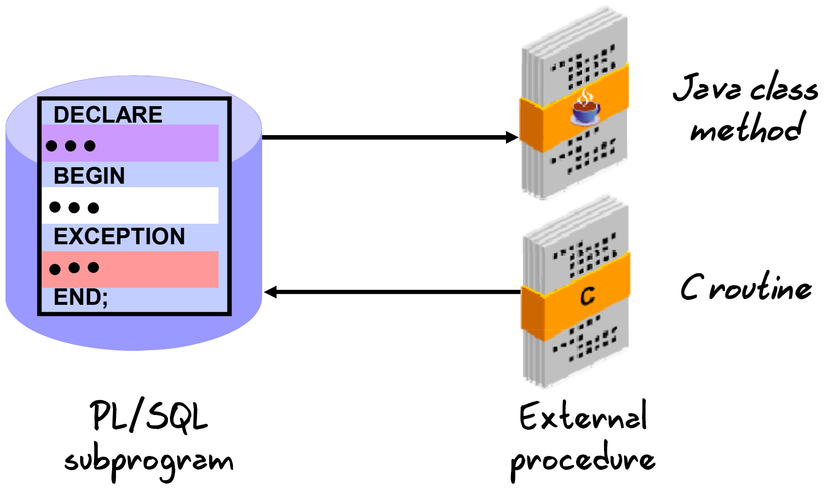
After completing this lesson, you should be able to execute the following:

- External C programs from PL/SQL
- Java programs from PL/SQL

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.



An *external procedure* (also called an *external routine*) is a routine stored in a dynamic link library (DLL), shared object (`.so` file in Linux or 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.

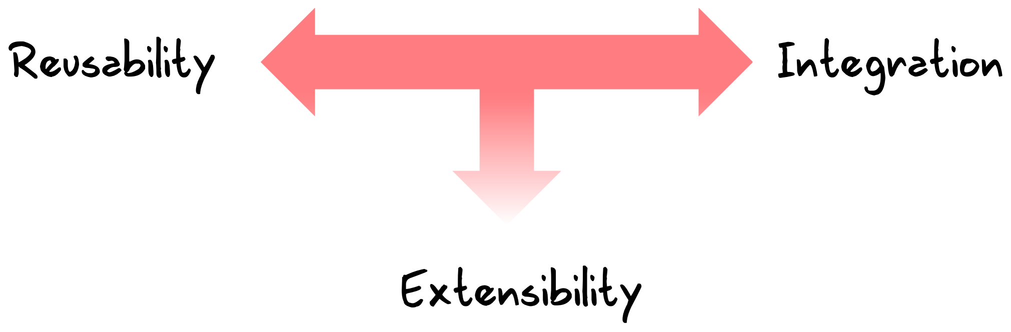
A *callout* is a call to the external procedure from your PL/SQL code.

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



The benefits of external procedures:

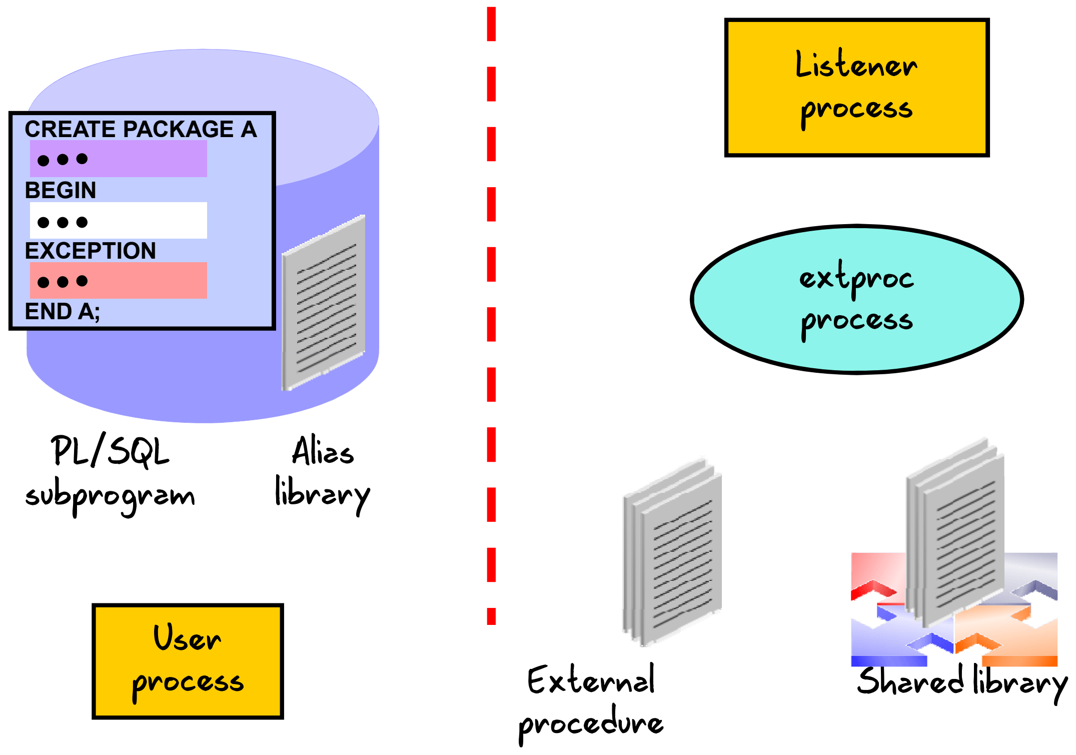
- External procedures integrate the strength and capability of different languages to give transparent access to these routines within the database.
- They provide functionality in the database that is specific to a particular application, company, or technological area.
- They can be shared by all users on a database, and they can be moved to other databases or computers, thereby providing standard functionality with limited cost in development, maintenance, and deployment.

Using an external procedure, you can invoke an external routine through PL/SQL. By using external procedures, 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.

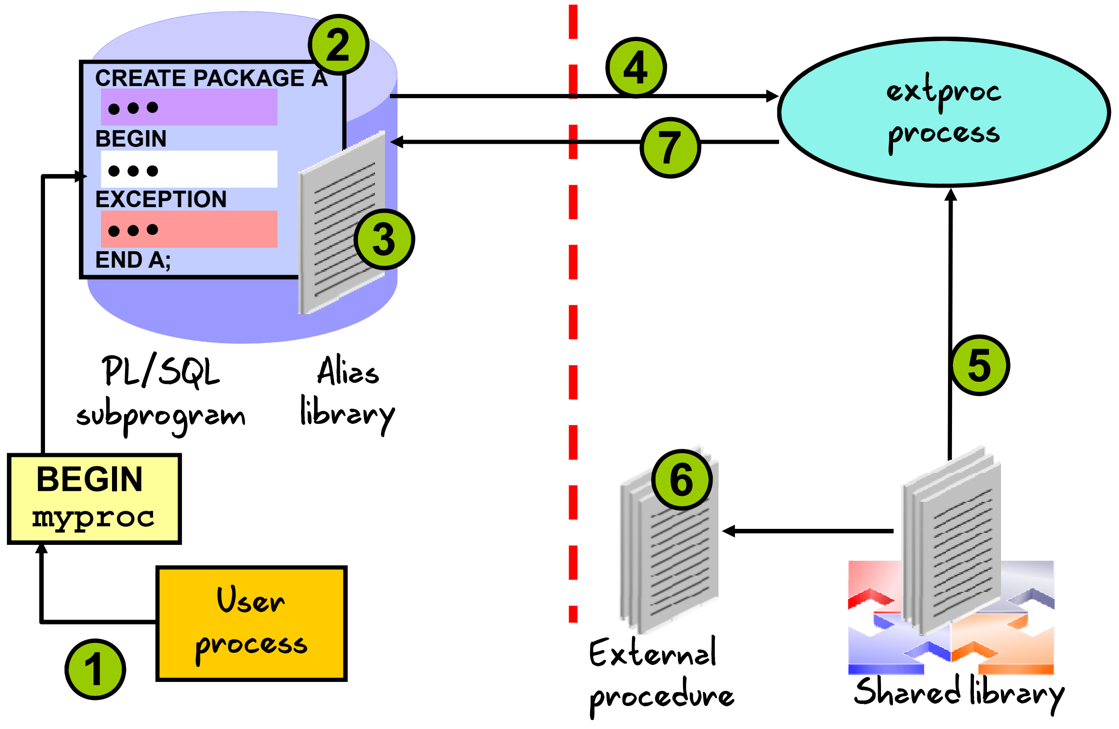
Example: A company has very complicated statistics programs written in C. The customer wants to access the data stored in an Oracle database and pass the data into the C programs. After the execution of the C programs, depending on the result of the evaluations, data is inserted into the appropriate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

External C Procedure Components



- **External 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

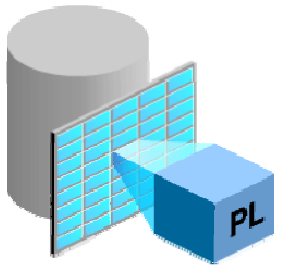
How PL/SQL Calls a C External Procedure



1. The user process invokes a PL/SQL program.
2. The server process executes a PL/SQL subprogram.
3. PL/SQL subprogram looks up the alias library.
4. Oracle Database starts the external procedure agent, `extproc`.
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.

`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 `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

Note: In releases before Oracle Database 11*g*, Oracle Listener spawned the multithreaded `extproc` agent, and you defined environment variables for `extproc` in the `listener.ora` file. Starting from Oracle Database 11*g*, by default, Oracle Database spawns `extproc` directly, eliminating the risk that Oracle Listener might spawn `extproc` unexpectedly.

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.

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

Development Steps for External C Procedures

1. and 2. These steps v*ary for each operating system; consult the documentation.*
3. Use the `CREATE LIBRARY` statement to create an alias library object.

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

4. Grant the `EXECUTE` privilege on the alias library.

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

An alias library is a database object that is used to map to an external shared library. An external procedure that you want to use must be stored in a DLL or a 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 must 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 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`.

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

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

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

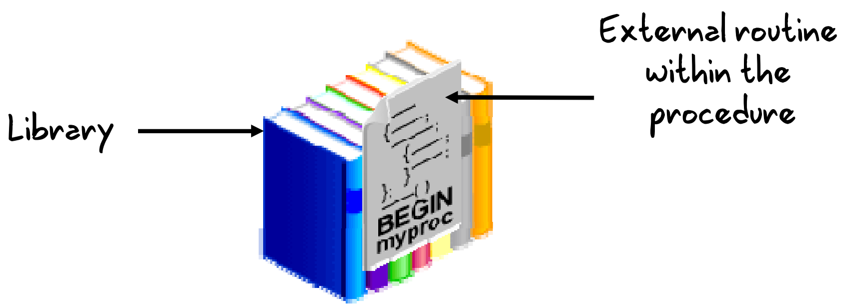
Dictionary Information

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.



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:

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

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

Call Specification

Call specification enables:

- **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**

The current way to publish external procedures is through call specifications. Call specifications enable 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 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 have the `RESTRICT_REFERENCES` pragma, use the `TRUST` option. The SQL engine cannot analyze those functions to determine whether they are free from side effects. The `TRUST` option makes it easier to call the Java and C procedures.

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]);
```

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`	Is the language in which the external routine was written (defaults to C)
	`LIBRARY` *`libname`*	Is the 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 is passed to the external routine for callbacks
	`parameters`	Identifies arguments passed to the external routine

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

Syntax Definitions

where:	`formal_parameter_name [INDICATOR]`	Is the 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 return value
	`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`	Is 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 `calc_tax` from a PL/SQL function:

```
CREATE OR REPLACE FUNCTION tax_amt (
 x BINARY_INTEGER)
 RETURN BINARY_INTEGER
  AS LANGUAGE C
  LIBRARY sys.c_utility
  NAME "calc_tax";
/
```

- The C prototype:

```
int calc_tax (n);
```

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

```
        int calc_tax (n);
```

To publish the `calc_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. `LIBRARY` identifies the library object that locates the C file. The `PARAMETERS` clause is not needed in this example, because the mapping of parameters is done on a positional basis.

Here is another example:

```
CREATE OR REPLACE PACKAGE Demo_pack
AUTHID DEFINER  AS
 PROCEDURE plsToC_demoExternal_proc (x PLS_INTEGER, y VARCHAR2, z DATE)
   AS LANGUAGE C
      NAME "C_demoExternal"
      LIBRARY SomeLib
      WITH CONTEXT
      PARAMETERS(CONTEXT, x INT, y STRING, z OCIDATE);
END;
```

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.

Here is a simple example of invoking the external routine:

```
BEGIN
  DBMS_OUTPUT.PUT_LINE(tax_amt(100));
END;
```

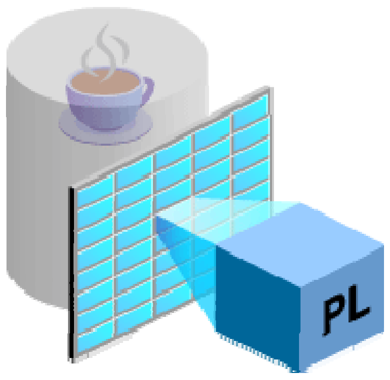
You can call the function in a cursor `FOR` loop or in any location where a PL/SQL function call is allowed:

```
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;
```

Working with Java Methods Using PL/SQL: Overview

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

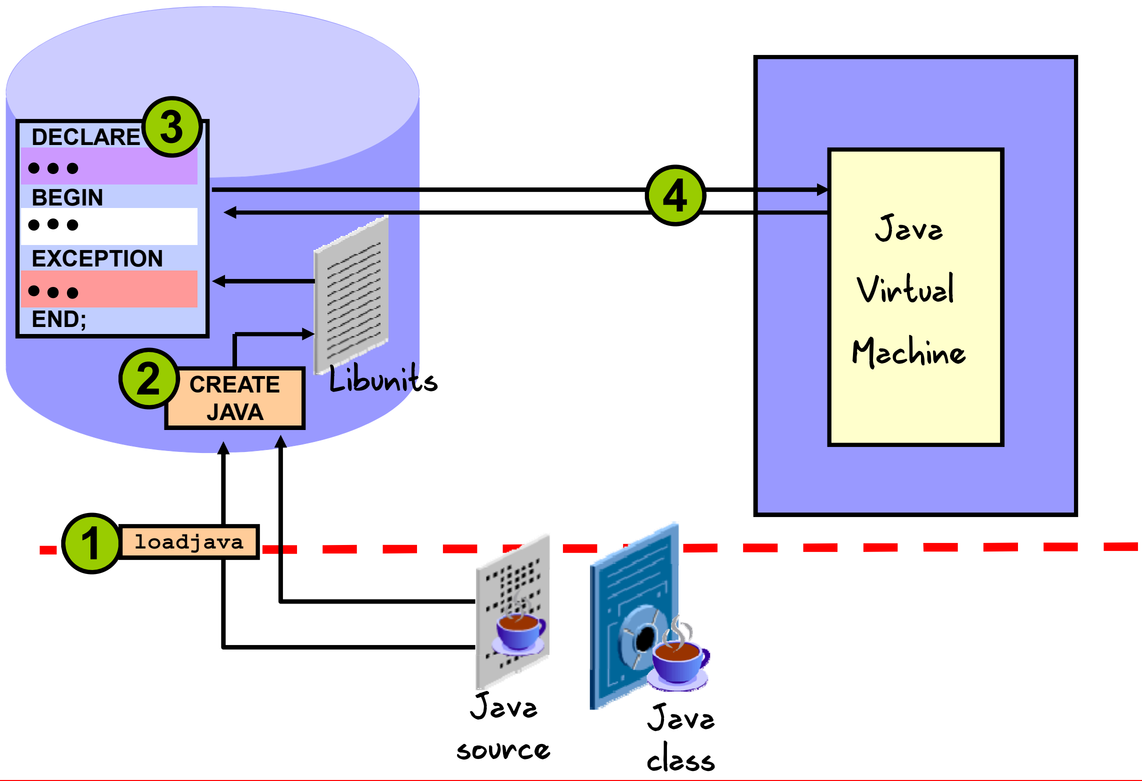
- Are stored in the database as procedures, functions, or triggers
- Run inside the database
- Manipulate data



The Oracle database can store Java classes (`.class` files) and Java source code (`.java` files), as procedures, functions, or triggers. These classes can manipulate data but cannot display GUI elements, such as Abstract Window Toolkit (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.

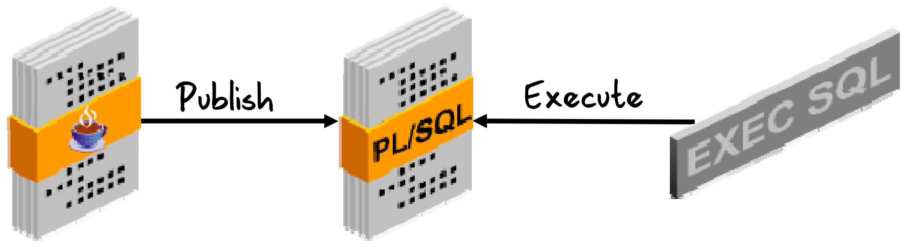
Calling a Java Class Method by Using PL/SQL



1. Use the `loadjava` utility from command line or from an application to upload the Java binaries and resources into a system-generated database table, where they are stored as Java schema objects.
2. The `loadjava` command-line utility uses the SQL `CREATE JAVA` statements to load Java source, class, or resource files into the RDBMS libunits. Libunits can be considered analogous to the DLLs written in C, although they map one-to-one with Java classes, whereas DLLs can contain multiple routines. Alternatively, you can implicitly call `CREATE JAVA` from SQL*Plus.
3. When you load a Java class into the database, its methods are not published automatically, because Oracle Database does not know which methods are safe entry points for calls from SQL. To publish the Java class method, create the PL/SQL subprogram unit specification that references the Java class methods.
4. Execute the PL/SQL subprogram that invokes the Java class method.

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.



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 level, use the `loadjava` command-line utility to load either the Java class file or the Java source file.

- To load the Java source file, use:

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

- To load the Java class file, use:

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

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

Java classes and their methods are stored in RDBMS libunits where the Java sources, binaries, and resources can be loaded.

Use the `loadjava` command-line utility to load and resolve the Java classes. Using the `loadjava` utility, you can upload the Java source, class, or 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
Factorial                      JAVA SOURCE

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

1. 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]';
```

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) ;
  }
}
```

You want to publish a Java method named `calcFactorial` that returns the factorial of its argument, as shown in the slide:

- The PL/SQL function `plstojavafac_fun` is created to identify the parameters and the Java characteristics.
- The `NAME` clause string uniquely identifies the Java method.
- The parameter named `N` corresponds to the `int` argument.

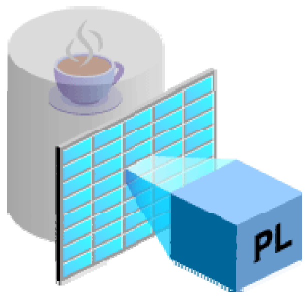
You can publish the Java method directly from the JVM.

Here is another example:

```
create or replace function dept_sal(p_deptno VARCHAR2)
return VARCHAR2
AS
LANGUAGE JAVA
NAME 'dept.sal(java.lang.String) return java.lang.String';
/
```

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.



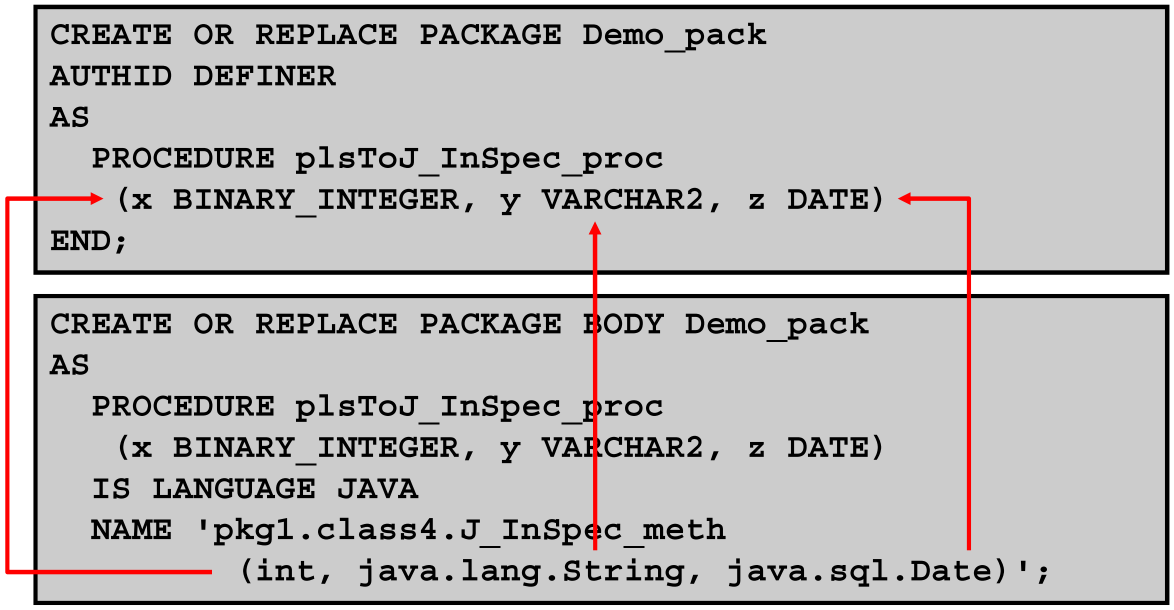
You can call the `calcFactorial` class method by using the following command:

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

Anonymous block completed
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)';
```

The examples in the slide create a package specification and body named `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.

Quiz

Which of the following statements is true about temporary `extproc`?

a. Oracle Database starts the external procedure agent, `extproc`.
b. The `extproc` process loads the shared library.
c. The `extproc` process links the server to the external file and executes the external procedure.
d. All of the above

Answer: d

Quiz

Call specifications do not:

a. Link the server to the external file and execute the external procedure
b. Dispatch the appropriate C or Java target procedure
c. Perform data type conversions
d. Perform parameter mode mappings
e. Perform automatic memory allocation and cleanup
f. Call Java methods or C procedures from database triggers

Answer: a

Quiz

Select the correct order of the steps required to execute a Java class method from PL/SQL:

A. Publish the Java class method by creating the PL/SQL subprogram unit specification that references the Java class methods.

B. Upload the Java file by using the `loadjava` command-line utility.

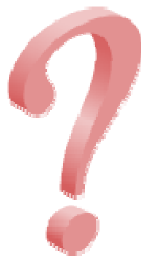
C. Execute the PL/SQL subprogram that invokes the Java class method.

a. A, B, C

b. B, A, C

c. C, A, B

d. C, B, A



Answer: b

Summary

In this lesson, you should have learned how to use:

- External C routines and call them from your PL/SQL programs
- Java methods and call them from your PL/SQL programs

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 7: Overview

This practice covers writing programs to interact with:

- C routines
- Java code

In this practice, you write two PL/SQL programs: one program calls an external C routine and the other calls a Java routine.

Use the `OE` schema for this practice.

8 Performance and Tuning

Objectives

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

- Describe and influence the compiler
- Tune PL/SQL code
- Enable intraunit inlining

In this lesson, the performance and tuning topics are divided into three main groups:

- Native and interpreted compilation
- Tuning PL/SQL code
- Intraunit inlining

In the compilation section, you learn about 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; and about data types and constraint issues.

With inlining, the compiler reviews code to see whether it can be inlined rather than referenced. You can influence the inlining process.

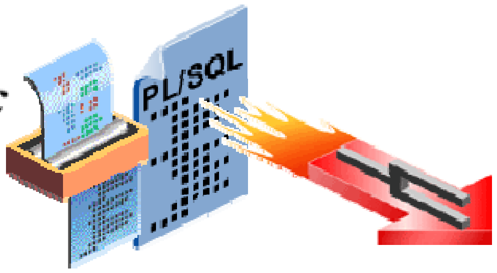
Lesson Agenda

- Using native and interpreted compilation methods
- Tuning PL/SQL code
- Enabling intraunit inlining

Native and Interpreted Compilation

Two compilation methods:

- Interpreted compilation
  - Default compilation method
  - Interpreted at run time
- Native compilation
  - Compiles into native code
  - Stored in the `SYSTEM` tablespace



You can compile your PL/SQL code by using either native compilation or interpreted compilation.

With interpreted compilation, the PL/SQL statements in a PL/SQL program unit are compiled into an intermediate form, machine-readable code, which is stored in the database dictionary and interpreted at run time. You can use PL/SQL debugging tools on program units compiled for interpreted mode.

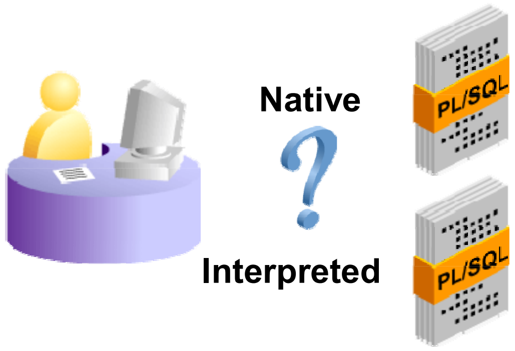
With PL/SQL native compilation, the PL/SQL statements in a PL/SQL program unit are compiled into native code and stored in the `SYSTEM` tablespace. Because the native code does not have to be interpreted at run time, it runs faster.

Native compilation applies only to PL/SQL statements. If your PL/SQL program contains only calls to SQL statements, it may not run faster when natively compiled; however, it will run at least as fast as the corresponding interpreted code. The compiled code and the interpreted code make the same library calls, so their action is the same.

The first time a natively compiled PL/SQL program unit is executed, it is fetched from the `SYSTEM` tablespace into the shared memory. Regardless of how many sessions call the program unit, the shared memory has only one copy of it. If a program unit is not being used, the shared memory it is using might be freed to reduce the memory load.

Deciding on a Compilation Method

- Use the interpreted mode when (typically during development):
  - You use a debugging tool, such as SQL Developer
  - You need the code compiled quickly
- Use the native mode when (typically post development):
  - Your code is heavily PL/SQL based
  - You want increased performance in production



When deciding on a compilation method, you need to examine:

- Where you are in the development cycle
- What the program unit does

If you are debugging and recompiling program units frequently, the interpreted mode has the following advantages:

- You can use PL/SQL debugging tools on program units compiled for interpreted mode (but not for those compiled for native mode).
- Compiling for interpreted mode is faster than compiling for native mode.

After completing the debugging phase of development, consider the following when determining whether to compile a PL/SQL program unit for native mode:

- The native mode provides the greatest performance gains for computation-intensive procedural operations. Examples are data warehouse applications and applications with extensive server-side transformations of data for display.
- The native mode provides the least performance gains for PL/SQL subprograms that spend most of their time executing SQL.
- When many program units (typically over 15,000) are compiled for native execution, and are simultaneously active, the large amount of shared memory required might affect system performance.

Setting the Compilation Method

- `PLSQL_CODE_TYPE`: Specifies the compilation mode for the PL/SQL library units

```
PLSQL_CODE_TYPE = { INTERPRETED | NATIVE }
```

- `PLSQL_OPTIMIZE_LEVEL`: Specifies the optimization level to be used to compile the PL/SQL library units

```
PLSQL_OPTIMIZE_LEVEL = { 0 | 1 | 2 | 3}
```

- In general, for faster performance, use the following setting:

```
PLSQL_CODE_TYPE = NATIVE
PLSQL_OPTIMIZE_LEVEL = 2
```

The `PLSQL_CODE_TYPE` Parameter

The `PLSQL_CODE_TYPE` compilation parameter determines whether the PL/SQL code is natively compiled or interpreted.

If you choose `INTERPRETED`:

- PL/SQL library units are compiled to PL/SQL bytecode format
- These modules are executed by the PL/SQL interpreter engine

If you choose `NATIVE`:

- PL/SQL library units (with the possible exception of top-level anonymous PL/SQL blocks) are compiled to native (machine) code
- Such modules are executed natively without incurring interpreter overhead

When the value of this parameter is changed, it has no effect on the PL/SQL library units that have already been compiled. The value of this parameter is stored persistently with each library unit. If a PL/SQL library unit is compiled natively, all subsequent automatic recompilations of that library unit use the native compilation. In Oracle Database 12*c*, native compilation is easier and more integrated, with fewer initialization parameters to set.

The `PLSQL_OPTIMIZE_LEVEL` Parameter

This parameter specifies the optimization level that is used to compile the PL/SQL library units. The higher the setting of this parameter, the more effort the compiler makes to optimize the PL/SQL library units. The available values are `0`, `1`, `2`, and `3`:

- **`0`:** Maintains the evaluation order and, therefore, the pattern of side effects, exceptions, and package initializations of Oracle9*i* and earlier releases. It also removes the new semantic identity of `BINARY_INTEGER` and `PLS_INTEGER`, and restores the earlier rules for the evaluation of integer expressions. Although the code runs somewhat faster than it did in Oracle9*i*, the use of level 0 forfeits most of the performance gains of PL/SQL achieved with Oracle Database 10*g* and later releases.
- **`1`:** Applies a wide range of optimizations to PL/SQL programs, including the elimination of unnecessary computations and exceptions, but generally does not move source code out of its original source order
- **`2`:** Applies a wide range of modern optimization techniques beyond those of level 1, including changes that may move source code relatively far from its original location
- **`3`:** Is available in Oracle Database 12*c.* It applies a wide range of optimization techniques beyond those of level 2, automatically including techniques not specifically requested. This enables procedure inlining, which is an optimization process that replaces procedure calls with a copy of the body of the procedure to be called. The copied procedure almost always runs faster than the original call. To allow subprogram inlining, either accept the default value of the `PLSQL_OPTIMIZE_LEVEL` initialization parameter (which is `2`) or set it to `3`. With `PLSQL_OPTIMIZE_LEVEL` = `2`, you must specify each subprogram to be inlined. With `PLSQL_OPTIMIZE_LEVEL` = `3`, the PL/SQL compiler seeks opportunities to inline subprograms beyond those that you specify.

Generally, setting this parameter to `2` pays off in terms of better execution performance. If, however, the compiler runs slowly on a particular source module or if optimization does not make sense for some reason (for example, during rapid turnaround development), setting this parameter to `1` results in almost as good a compilation with less use of compile-time resources. The value of this parameter is stored persistently with the library unit.

Viewing the Compilation Settings

```
DESCRIBE ALL_PLSQL_OBJECT_SETTINGS
```

Displays the settings for a PL/SQL object

```
 Name                         Null?    Type
 ------------------------- -------- --------------------
 OWNER                        NOT NULL VARCHAR2(30)
 NAME                         NOT NULL VARCHAR2(30)
 TYPE                                  VARCHAR2(12)
 PLSQL_OPTIMIZE_LEVEL                  NUMBER
 PLSQL_CODE_TYPE                       VARCHAR2(4000)
 PLSQL_DEBUG                           VARCHAR2(4000)
 PLSQL_WARNINGS                        VARCHAR2(4000)
 NLS_LENGTH_SEMANTICS                  VARCHAR2(4000)
 PLSQL_CCFLAGS                         VARCHAR2(4000)
 PLSCOPE_SETTINGS                      VARCHAR2(4000)
```

The `USER|ALL|DBA_PLSQL_OBJECT_SETTINGS` data dictionary views are used to display the settings for a PL/SQL object.

The columns of the `USER_PLSQL_OBJECTS_SETTINGS` data dictionary view include:

- **Owner:** The owner of the object. This column is not displayed in the `USER_PLSQL_OBJECTS_SETTINGS` view.
- **Name:** The name of the object
- **Type:** The available choices are `PROCEDURE`, `FUNCTION`, `PACKAGE`, `PACKAGE BODY`, `TRIGGER`, `TYPE`, or `TYPE BODY`.
- **`PLSQL_OPTIMIZE_LEVEL`:** The optimization level that was used to compile the object
- **`PLSQL_CODE_TYPE`:** The compilation mode for the object
- **`PLSQL_DEBUG`:** Specifies whether or not the object was compiled for debugging
- **`PLSQL_WARNINGS`:** The compiler warning settings used to compile the object
- **`NLS_LENGTH_SEMANTICS`:** The national language support (NLS) length semantics used to compile the object
- **`PLSQL_CCFLAGS`:** The conditional compilation flag used to compile the object
- **`PLSCOPE_SETTINGS`:** Controls the compile time collection, cross reference, and storage of PL/SQL source code identifier data

Viewing the Compilation Settings

```
SELECT name, plsql_code_type, plsql_optimize_level
FROM   user_plsql_object_settings;
```

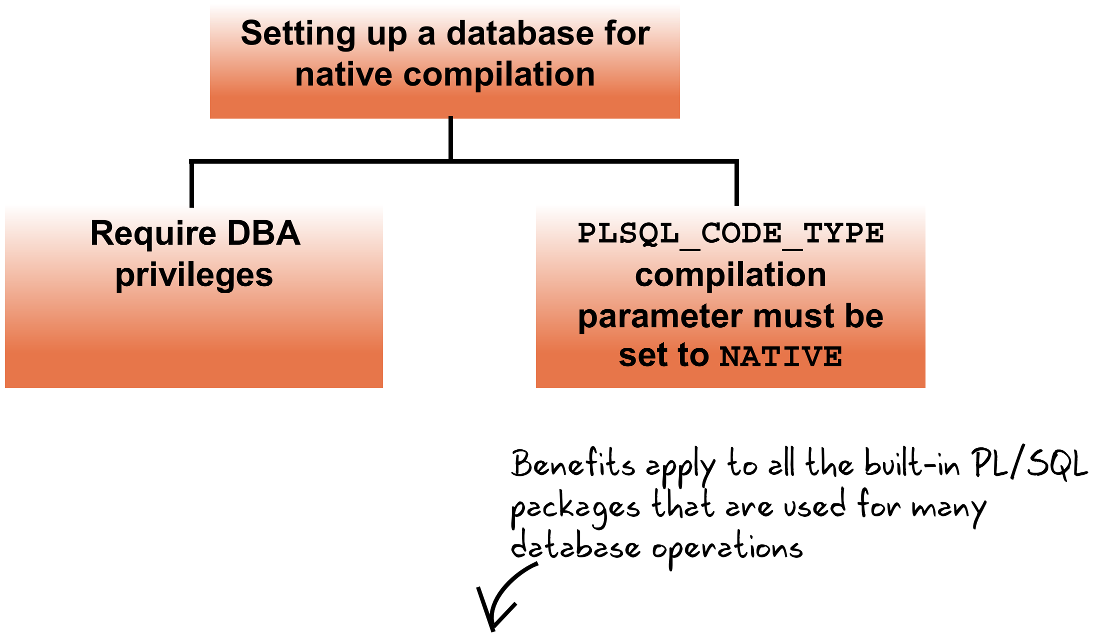
To view the compilation settings

```
NAME                  PLSQL_CODE_TYP PLSQL_OPTIMIZE_LEVEL
-------------------- -------------- --------------------
ACTIONS_T              INTERPRETED                      2
ACTION_T               INTERPRETED                      2
ACTION_V               INTERPRETED                      2
ADD_ORDER_ITEMS        INTERPRETED                      2
CATALOG_TYP            INTERPRETED                      2
CATALOG_TYP            INTERPRETED                      2
CATALOG_TYP            INTERPRETED                      2
CATEGORY_TYP           INTERPRETED                      2
CATEGORY_TYP           INTERPRETED                      2
COMPOSITE_CATEGORY_TYP INTERPRETED                      2
...
```

Set the values of the compiler initialization parameters by using the `ALTER SYSTEM` or `ALTER SESSION` statements.

The parameters' values are accessed when the `CREATE OR REPLACE` or `ALTER` statements are executed.

Setting Up a Database for Native Compilation


Setting up a database for native compilation
Require DBA privileges
PLSQL_CODE_TYPE compilation parameter must be set to NATIVE
Benefits apply to all the built-in PL/SQL packages that are used for many database operations

```
ALTER SYSTEM SET PLSQL_CODE_TYPE = NATIVE;
```

If you have DBA privileges, you can set up a new database for PL/SQL native compilation by setting the `PLSQL_CODE_TYPE` compilation parameter to `NATIVE`. The performance benefits apply to all built-in PL/SQL packages that are used for many database operations.

Compiling a Program Unit for Native Compilation

```
SELECT name, plsql_code_type, plsql_optimize_level      1
FROM   user_plsql_object_settings
WHERE  name = 'ADD_ORDER_ITEMS';

NAME                   PLSQL_CODE_T PLSQL_OPTIMIZE_LEVEL
---------------------- ------------ --------------------
ADD_ORDER_ITEMS        INTERPRETED                     2
ALTER SESSION SET PLSQL_CODE_TYPE = 'NATIVE';      2

ALTER PROCEDURE add_order_items COMPILE;           3

SELECT name, plsql_code_type, plsql_optimize_level      4
FROM   user_plsql_object_settings
WHERE  name = 'ADD_ORDER_ITEMS';

NAME                   PLSQL_CODE_T PLSQL_OPTIMIZE_LEVEL
---------------------- ------------ --------------------
ADD_ORDER_ITEMS        NATIVE                          2
```

To change a compiled PL/SQL object from interpreted code type to native code type, you must set the `PLSQL_CODE_TYPE` parameter to `NATIVE` (optionally, set the other parameters), and then recompile the program.

In the example in the slide:

1. The compilation type is checked on the `ADD_ORDER_ITEMS` program unit.
2. The compilation method is set to `NATIVE` at the session level.
3. The `ADD_ORDER_ITEMS` program unit is recompiled.
4. The compilation type is checked again on the `ADD_ORDER_ITEMS` program unit to verify that it changed.

If you want to compile an entire database for native or interpreted compilation, scripts are provided to help you do so.

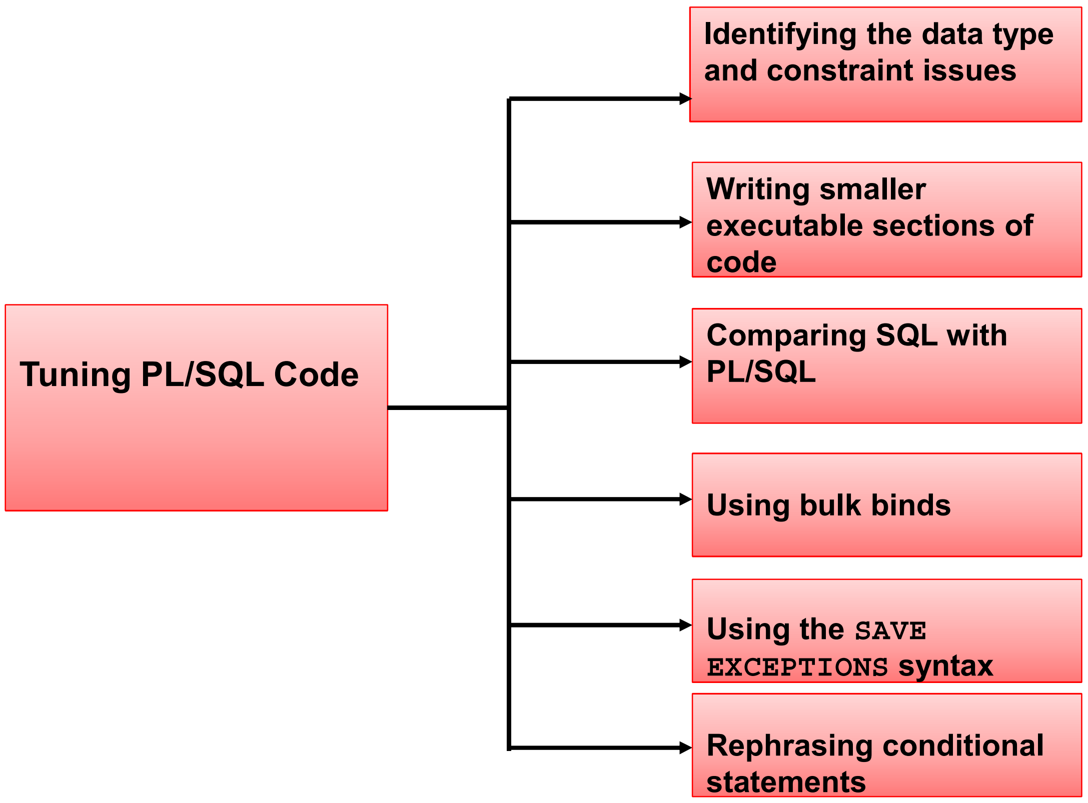
- You must have DBA privileges.
- Set `PLSQL_CODE_TYPE` at the system level.
- Run the `dbmsupgnv.sql`-supplied script that is found in the `\Oraclehome\product\12.1.0\db_1\RDBMS\ADMIN` folder.

For detailed information, see the *Oracle Database PL/SQL Language Reference 12c* reference manual.

Lesson Agenda

- Using native and interpreted compilation methods
- Tuning PL/SQL code
- Enabling intraunit inlining

Tuning PL/SQL Code


Tuning PL/SQL Code
Identifying the data type and constraint issues
Writing smaller executable sections of code
Comparing SQL with PL/SQL
Using bulk binds
Using the SAVE EXCEPTIONS syntax
Rephrasing conditional statements

By tuning your PL/SQL code, you can customize its performance to best meet your needs.

You can tune your PL/SQL code by:

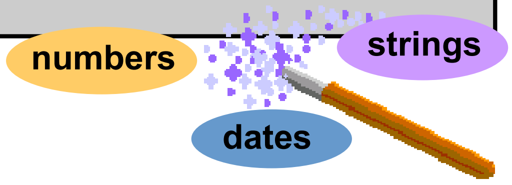
- Identifying the data type and constraint issues
  - Data type conversion
  - The `NOT NULL` constraint
  - `PLS_INTEGER`
  - `SIMPLE_INTEGER`
- Writing smaller executable sections of code
- Comparing SQL with PL/SQL
- Understanding how bulk binds can improve performance
- Using the FORALL support with bulk binding
- Handling and saving exceptions with the `SAVE EXCEPTIONS` syntax
- Rephrasing conditional statements

In the following slides, you learn about some of the main PL/SQL tuning issues that can improve the performance of your PL/SQL applications.

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;
```



At run time, PL/SQL automatically performs implicit conversions between structurally different data types. By avoiding implicit conversions, you can improve the performance of your code. The following are the major problems with implicit data type conversion:

- It is nonintuitive and can result in unexpected results.
- You have no control over the implicit conversion.

In the example in the slide, 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, use the following built-in functions:

- `TO_DATE`
- `TO_NUMBER`
- `TO_CHAR`
- `CAST`

Understanding the `NOT NULL` Constraint

```
PROCEDURE calc_m IS
m NUMBER NOT NULL:=0;
a NUMBER;
b NUMBER;
BEGIN
  m := a + b;
END;
```

The value of the expression a + b is assigned to a temporary variable, which is then tested for nullity.

```
PROCEDURE calc_m IS
m NUMBER; --no constraint
...
BEGIN
m := a + b;
IF m IS NULL THEN
     -- raise error
END IF;
END;
```

This is a better way to check nullity; no performance overhead.

In PL/SQL, using the `NOT NULL` constraint incurs a small performance cost. Therefore, use it with care. Consider the first example 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 in the bottom half of the slide.

Note that the subtypes `NATURALN` and `POSTIVEN` are defined as the `NOT NULL` subtypes of `NATURAL` and `POSITIVE`. Using them incurs the same performance cost as seen in the slide's first example.

Working of the `NOT NULL` Constraint

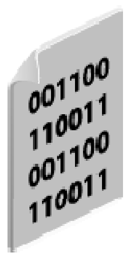
Using the `NOT NULL` Constraint	Not Using the Constraint
Slower	Faster
No extra coding is needed.	It requires extra coding, which is error prone.
When an error is implicitly raised, the value of m is preserved.	When an error is explicitly raised, the old value of m is lost.

The comparison of using the `NOT NULL` constraint and not using the constraint is shown in this slide.

Using the `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.



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 `BINARY_FLOAT` and `BINARY_DOUBLE` data types are also faster than the `NUMBER` data type.

Using the `SIMPLE_INTEGER` Data Type

- Definition:
  - Is a predefined subtype
  - Has the range –2147483648 .. 2147483648
  - Does not include a null value
  - Is allowed anywhere in PL/SQL where the `PLS_INTEGER` data type is allowed
- Benefits:
  - Eliminates the overhead of overflow checking
  - Is estimated to be 2–10 times faster when compared with the `PLS_INTEGER` type with native PL/SQL compilation

The `SIMPLE_INTEGER` data type is a predefined subtype of the `BINARY_INTEGER` (or `PLS_INTEGER`) data type that has the same numeric range as `BINARY_INTEGER`. It differs significantly from `PLS_INTEGER` in its overflow semantics. Incrementing the largest `SIMPLE_INTEGER` value by one silently produces the smallest value, and decrementing the smallest value by one silently produces the largest value. These “wrap around” semantics conform to the Institute of Electrical and Electronics Engineers (IEEE) standard for 32-bit integer arithmetic.

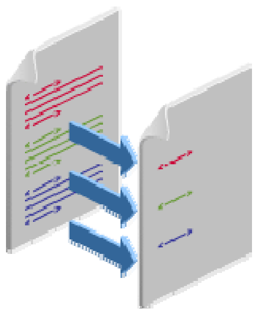
The key features of the `SIMPLE_INTEGER` predefined subtype are the following:

- Includes the range of –2147483648.. +2147483648
- Has a `NOT NULL` constraint
- Wraps rather than overflows
- Is faster than `PLS_INTEGER`

Without the overhead of overflow checking and nullness checking, the `SIMPLE_INTEGER` data type provides significantly better performance than `PLS_INTEGER` when the `PLSQL_CODE_TYPE` parameter is set to `native`, because arithmetic operations on the former are performed directly in the machine’s hardware. The performance difference is less noticeable when the `PLSQL_CODE_TYPE` parameter is set to `interpreted`; however, even with this setting, the `SIMPLE_INTEGER` type is faster than the `PLS_INTEGER` type.

Modularizing Your Code

- Limit the number of lines of code between a `BEGIN` and an `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.

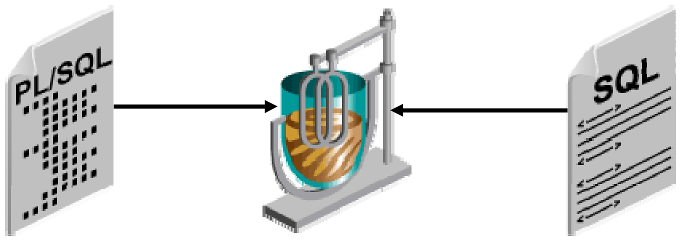


By writing smaller sections of executable code, you can make the code easier to read, understand, and maintain. When developing an application, use a stepwise refinement. Make a general description of what you want your program to do, and then implement the details in subroutines. Using local modules and packaged programs can help keep each executable section small. This makes 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



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 multiple-row queries.

Though 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;...
```

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 that are 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. It is also better to use the hint APPEND, which instructs the optimizer to use direct-path insert for a faster performance. Including the `LOG ERRORS` clause enables the insert operations to complete, regardless of errors.

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 by 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;
...
```

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);
```

From a design standpoint, do not embed your SQL statements directly within the application code. It is better if you write procedures to perform your SQL statements.

Pros

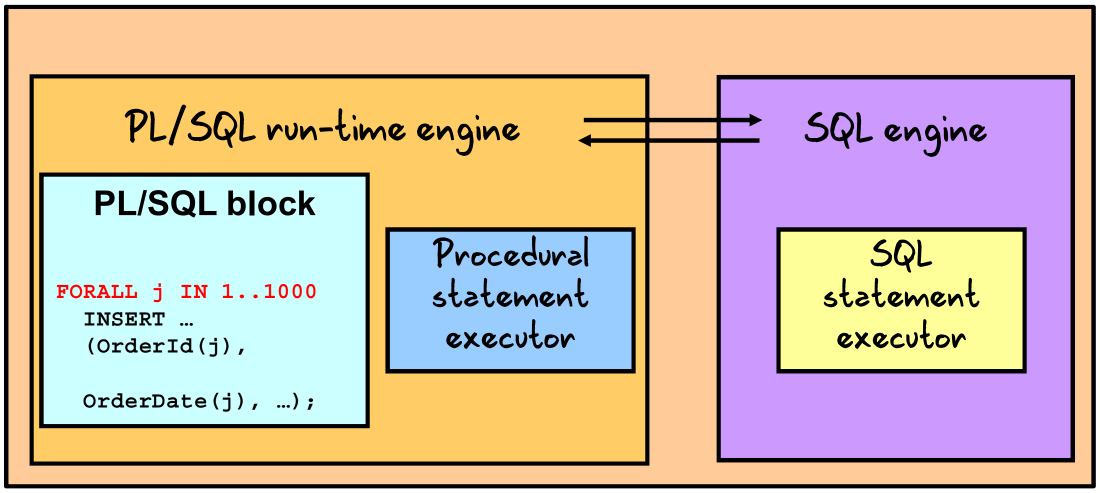
- If you design your application so that all programs that perform an insert on a specific table use the same `INSERT` statement, your application will run faster because of less parsing and reduced demands on the System Global Area (SGA) memory.
- Your program will also handle data manipulation language (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.



```
FORALL j IN 1..1000
  INSERT …
  (OrderId(j),

  OrderDate(j), …);
```

With bulk binds, you can improve performance by decreasing the number of context switches between the SQL and PL/SQL engines. When a PL/SQL program executes, each time a SQL statement is encountered, there is a switch between the PL/SQL engine and the SQL engine. The more the number of switches, the less the efficiency.

Improved Performance

Bulk binding enables you to implement array fetching. With bulk binding, entire collections, not just individual elements, are passed back and forth. Bulk binding can be used with nested tables, varrays, and associative arrays.

The more the rows affected by a SQL statement, the greater is the performance gain with bulk binding.

Using Bulk Binding

Bind whole arrays of values simultaneously, 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(...);
...
```

In the first example shown in the slide, one row at a time is inserted into the target table. In the second example, the `FOR` loop is changed to a `FORALL` (which has an implicit loop) and all the immediately subsequent DML statements are processed in bulk. The entire code examples, along with the timing statistics for running each `FOR` loop example, are as follows.

First, create the demonstration table:

```
CREATE TABLE bulk_bind_example_tbl (
  num_col NUMBER,
  date_col DATE,
  char_col VARCHAR2(40));
```

Then, set the SQL*Plus `TIMING` variable on. Setting it on enables you to see the approximate elapsed time of the last SQL statement:

```
SET TIMING ON
```

Then, 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;
```

```
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;
/

2.184ms elapsed
```

Finally, run this block of code that includes a `FORALL` loop to insert 50,000 rows. Note the significant decrease in the timing when using `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;
/

828ms elapsed
```

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;
```

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 runs 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;
  ...
```

You can also use the LIMIT option with BULK COLLECT. This gives you control over the amount of processed rows in one step.

```
FETCH cv_customers BULK COLLECT
  INTO v_custnos, v_last_name, v_email
  LIMIT 200;
```

Using Bulk Binding

Use the `RETURNING` clause to retrieve information about the rows that are 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;
```

Often, applications need information about the row that is affected by a SQL operation; for example, to generate a report or take action. Using the `RETURNING` clause, you can retrieve information about the rows that you 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 about 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 in the slide, 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;
```

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 Developer 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` keyword 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.

To handle the exceptions encountered during a `BULK BIND` operation, you can add the keyword `SAVE EXCEPTIONS` to your `FORALL` statement. Without it, if a 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 `%BULK_EXCEPTIONS` cursor attribute, 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 `SAVE EXCEPTIONS` keyword, 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;
/
```

In the example in the slide, the `EXCEPTION_INIT` pragma defines an exception named `BULK_ERRORS` and associates the name with 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 the code tried to divide by zero three times. To get the Oracle error message (which includes the code), you pass `SQL%BULK_EXCEPTIONS(i).ERROR_CODE` to the error-reporting function `SQLERRM`.

Note: If the `FORALL` statement uses the `INDICES OF` clause to process a sparse collection, `%BULK_ROWCOUNT` has corresponding sparse subscripts. If the `FORALL` statement uses the `VALUES OF` clause to process a subset of elements, `%BULK_ROWCOUNT` has subscripts corresponding to the values of the elements in the index collection.

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
```

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;
```

In logical expressions, improve performance by carefully tuning conditional constructs.

When evaluating a logical expression, PL/SQL stops evaluating the expression as soon as the result is 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;
```

If you have a situation where you are checking a list of choices for a mutually exclusive result, use the `ELSIF` syntax, because it offers the most efficient implementation. With `ELSIF`, after a branch evaluates to `TRUE`, the other branches are not executed.

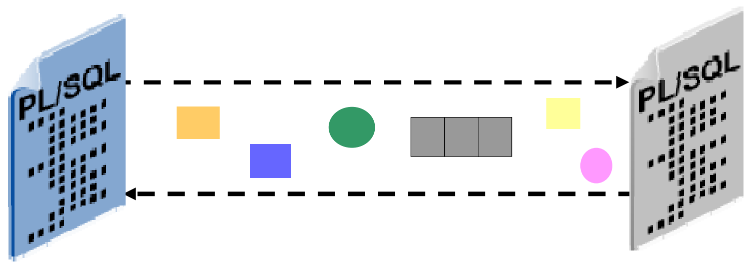
In the example shown on the right in the slide, every `IF` statement is executed. In the example on the left, after a branch is found to be true, the rest of the branch conditions are not evaluated.

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;
```

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 the `IN OUT` parameters.



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 is stored before the routine is executed. 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 not to make a backup copy of the parameter that is being passed. However, be careful when you use the `NOCOPY` compiler hint, because your results are not predictable if your program encounters an exception.

Passing Data Between PL/SQL Programs

Pass records as parameters to encapsulate data, and 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);
```

You can declare user-defined records as formal parameters of procedures and functions as shown in the slide. By using records to pass values, you are encapsulating the data being passed. This 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 `ORDER_TOTAL` field 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;
```

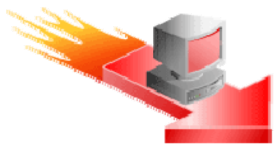
You can declare collections as formal parameters of procedures and functions. In the example in the slide, associative arrays are declared as the formal parameters of two packaged procedures. If you were to use scalar variables to pass the data, you would need to code and maintain many more declarations.

Lesson Agenda

- Using native and interpreted compilation methods
- Tuning PL/SQL code
- Enabling intraunit inlining

Intraunit Inlining: Introduction

- Definition:
  – Inlining is the replacement of a call to a subroutine with a copy of the body of the subroutine that is called.
  – The copied procedure generally runs faster than the original.
  – The PL/SQL compiler can automatically find the calls that should be inlined.
- Benefits:
  – When applied judiciously, inlining can provide large performance gains (by a factor of 2–10).



Procedure inlining is an optimization process that replaces procedure calls with a copy of the body of the procedure to be called. The copied procedure almost always runs faster than the original call, because:

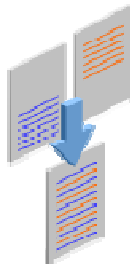
- The need to create and initialize the stack frame for the called procedure is eliminated
- The optimization can be applied over the combined text of the call context and the copied procedure body
- Propagation of constant actual arguments often causes the copied body to collapse under optimization

When inlining is achieved, you can see performance gains of 2–10 times.

With the recent releases, the PL/SQL compiler can automatically find calls that should be inlined, and can do the inlining correctly and quickly. There are some controls to specify where and when the compiler should do this work (using the `PLSQL_OPTIMIZATION_LEVEL` database parameter), but usually a general request is sufficient.

Using Inlining

- Implement inlining via two methods:
  – Oracle parameter `PLSQL_OPTIMIZE_LEVEL`
  – `PRAGMA INLINE`
- It is recommended that you inline:
  – Small programs
  – Programs that are frequently executed
- Use performance tools to identify the hotspots that are suitable for inline applications:
  – `plstimer`



When implementing inlining, it is recommended that the process be applied to smaller programs, and programs that execute frequently. For example, you may want to inline small helper programs.

To help you identify which programs to inline, you can use the `plstimer` PL/SQL performance tool. This tool specifically analyzes program performance in terms of the time spent in procedures and the time spent on particular call sites. It is important that you identify the procedure calls that may benefit from inlining.

There are two ways to use inlining:

1. Set the `PLSQL_OPTIMIZE_LEVEL` parameter to `3`. When this parameter is set to `3`, the PL/SQL compiler searches for calls that might profit from inlining and inlines the most profitable calls. Profitability is measured by those calls that help the program speed up the most and keep the compiled object program as short as possible.

```
ALTER SESSION SET plsql_optimize_level = 3;
```

2. Use `PRAGMA INLINE` in your PL/SQL code. This identifies whether a specific call should be inlined. Setting this pragma to “YES” has an effect only if the optimize level is set to two or higher.

Inlining Concepts

Noninlined program:

```
CREATE OR REPLACE PROCEDURE small_pgm
IS
  a NUMBER;
  b NUMBER;

  PROCEDURE touch(x IN OUT NUMBER, y NUMBER)
  IS
  BEGIN
    IF y > 0 THEN
      x := x*x;
    END IF;
  END;

BEGIN
  a := b;
  FOR I IN 1..10 LOOP
    touch(a, -17);
    a := a*b;
  END LOOP;
END small_pgm;
```

The example shown in the slide will be expanded to show you how a procedure is inlined.

The `a:=a*b` assignment at the end of the loop looks like it could be moved before the loop; however, it cannot, because `a` is passed as an `IN OUT` parameter to the `TOUCH` procedure. The compiler cannot be certain what the procedure does to its parameters. This results in the multiplication and in the assignment being completed 10 times instead of only once, even though multiple executions are not necessary.

Inlining Concepts

Examine the loop after inlining:

```
...
BEGIN
  a := b;
  FOR i IN 1..10 LOOP
    IF -17 > 0 THEN
      a := a*a;
    END IF;
    a := a*b;
  END LOOP;
END small_pgm;
...
```

The code in the slide shows what happens to the loop after inlining.

Inlining Concepts

The loop is transformed in several steps:

```
a := b;
 FOR i IN 1..10 LOOP ...
   IF false THEN
     a := a*a;
   END IF;
   a := a*b;
 END LOOP;

 a := b;
 FOR i IN 1..10 LOOP ...
   a := a*b;
 END LOOP;

 a := b;
 a := a*b;
 FOR i IN 1..10 LOOP ...
 END LOOP;

 a := b*b;
   FOR i IN 1..10 LOOP ...
 END LOOP;
```

Because the insides of the procedure are now visible to the compiler, it can transform the loop in several steps, as shown in the slide.

Instead of 11 assignments (one outside of the loop) and 10 multiplications, only one assignment and one multiplication are performed. If the loop ran a million times (instead of 10), the savings would be a million assignments. For code that contains deep loops that are executed frequently, inlining offers tremendous savings.

Inlining: Example

- Set the PLSQL_OPTIMIZE_LEVEL session-level parameter to a value of 2 or 3:

```
ALTER PROCEDURE small_pgm COMPILE
  PLSQL_OPTIMIZE_LEVEL = 3 REUSE SETTINGS;
```

  – Setting it to 2 means no automatic inlining is attempted.
  – Setting it to 3 means automatic inlining is attempted, but no pragmas are necessary.
- Within a PL/SQL subroutine, use PRAGMA INLINE:
  – NO means no inlining occurs regardless of the level and of the YES pragmas.
  – YES means inline at level 2 of a particular call and increase the priority of inlining at level 3 for the call.

To influence the optimizer to use inlining, you can set the PLSQL_OPTIMIZE_LEVEL parameter to a value of 2 or 3. By setting this parameter, you are making a request that inlining be used. It is up to the compiler to analyze the code and determine whether inlining is appropriate. When the optimize level is set to 3, the PL/SQL compiler searches for calls that might profit from inlining and inlines the most profitable calls.

In rare cases, if the overhead of the optimizer makes the compilation of very large applications take too long, you can lower the optimization by setting PLSQL_OPTIMIZE_LEVEL=1 instead of its default value of 2. In even rarer cases, you might see a change in exception action, either an exception that is not raised at all, or one that is raised earlier than expected. Setting PLSQL_OPTIMIZE_LEVEL=1 prevents the code from being rearranged.

To enable inlining within a PL/SQL subroutine, you can use PRAGMA INLINE to suggest that a specific call be inlined.

Inlining: Example

After setting the `PLSQL_OPTIMIZE_LEVEL` parameter, use a pragma:

```
CREATE OR REPLACE PROCEDURE small_pgm
IS
  a PLS_INTEGER;
  FUNCTION add_it(a PLS_INTEGER, b PLS_INTEGER)
  RETURN PLS_INTEGER
  IS
  BEGIN
    RETURN a + b;
  END;
BEGIN
  pragma INLINE (add_it, 'YES');
  a := add_it(3, 4) + 6;
END small_pgm;
```

Within a PL/SQL subroutine, you can use `PRAGMA INLINE` to suggest that a specific call be inlined. When using `PRAGMA INLINE`, the first argument is the simple name of a subroutine, a function name, a procedure name, or a method name. The second argument is either the constant string '`NO`' or '`YES`'. The pragma can go before any statement or declaration. If you put it in the wrong place, you receive a syntax error message from the compiler.

To identify that a specific call should not be inlined, use:

```
PRAGMA INLINE (function_name, 'NO');
```

Setting the `PRAGMA INLINE` to '`NO`' always works, regardless of any other pragmas that might also apply to the same statement. The pragma also applies at all optimization levels, and it applies no matter how badly the compiler would like to inline a particular call. If you are certain that you do not want some code inlined (perhaps due to the large size), you can set this to `NO`.

Setting the `PRAGMA INLINE` to '`YES`' strongly encourages the compiler to inline the call. The compiler keeps track of the resources used during inlining and makes the decision to stop inlining when the cost becomes too high.

If inlining is requested and you have the compiler warnings turned on, you see the message:

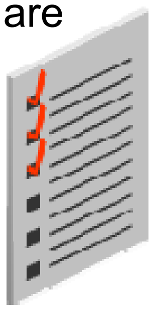
```
PLW-06004: inlining of call of procedure ADD_IT requested.
```

If inlining is applied, you see the compiler warning (it is more of a message):

```
PLW-06005: inlining of call of procedure 'ADD_IT' was done.
```

Inlining: Guidelines

- Pragmas apply only to calls in the next statement following the pragma.
- Programs that make use of smaller helper subroutines are good candidates for inlining.
- Only local subroutines can be inlined.
- You cannot inline an external subroutine.
- Inlining can increase the size of a unit.
- Be careful about suggesting to inline functions that are deterministic.



The compiler inlines code automatically, provided that you are using native compilation and have set the `PLSQL_OPTIMIZE_LEVEL` to `3`. If you have set `PLSQL_Warnings = 'enable:all'`, using the SQL*Plus `SHOW ERRORS` command displays the name of the code that is inlined.

- The `PLW-06004` compiler message tells you that a `pragma INLINE('YES')` referring to the named procedure was found. The compiler will, if possible, inline this call.
- The `PLW-06005` compiler message tells you the name of the code that is inlined.

Alternatively, you can query the `USER/ALL/DBA_ERRORS` dictionary view.

Deterministic functions compute the same outputs for the same inputs every time the functions are invoked, and have no side effects. The PL/SQL compiler can figure out whether a function is deterministic; it may not find all that truly are, but it finds many of them. It never mistakes a nondeterministic function for a deterministic function.

Quiz

Which of the following statements are true?

a. Use the native mode during development.

b. Because the native code does not have to be interpreted at run time, it runs faster.

c. The interpreted compilation is the default compilation method.

d. To change a compiled PL/SQL object from interpreted code type to native code type, you must set the `PLSQL_CODE_TYPE` parameter to `NATIVE`, and then recompile the program.

Answer: b, c, d

Quiz

You can tune your PL/SQL code by:

a. Writing longer executable sections of code
b. Avoiding bulk binds
c. Using the `FORALL` support with bulk binding
d. Handling and saving exceptions with the `SAVE EXCEPTIONS` syntax
e. Rephrasing conditional statements

Answer: c, d, e

Quiz

Which of the following statements are true with reference to inlining?

a. Pragmas apply only to calls in the next statement following the pragma.
b. Programs that make use of smaller helper subroutines are bad candidates for inlining.
c. Only local subroutines can be inlined.
d. You cannot inline an external subroutine.
e. Inlining can decrease the size of a unit.

Answer: a, c, d

Summary

In this lesson, you should have learned how to:

- Decide when to use native or interpreted compilation
- 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
- Identify opportunities for inlining PL/SQL code
- Use native compilation for faster PL/SQL execution

There are several methods that help you tune your PL/SQL application.

When tuning PL/SQL code, consider using the `RETURNING` clause and 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.

By using native compilation, you can benefit from performance gains for computation-intensive procedural operations.

Practice 8: Overview

This practice covers the following topics:

- Tuning PL/SQL code to improve performance
- Coding with bulk binds to improve performance

In this practice, you tune some of the code that you created for the `OE` application.

- Break a previously built subroutine into smaller executable sections
- Pass collections into subroutines
- Add error handling for `BULK INSERT`

Use the `OE` schema for this practice.