Oracle Database 10*g*: PL/SQL Fundamentals

Volume 1 • Student Guide

D17112GC21

Edition 2.1

December 2006

D48243

Authors

Tulika Srivastava
Sunitha Patel

Technical Contributors and Reviewers

Chaitanya Koratamaddi
Christoph Burandt
Zarko Cesljas
Dairy Chan
Isabelle Cornu
Kathryn Cunningham
Burt Demchick
Joel Goodman
Jonathan Grove
Jessie Ho
Craig Hollister
Alison Holloway
Bryn Llewellyn
Malika Marghadi
Hildegard Mayr
Nancy Greenberg
Miyuki Osato
Nagavalli Pataballa
Srinivas Putrevu
Bryan Roberts
Helen Robertson
Grant Spencer
Lex Van Der Werff

Editors

Richard Wallis
Arijit Ghosh

Graphic Designer

Steve Elwood

Publishers

Nita Brozowski
Srividya Rameshkumar

Contents

Preface

Profile

Before You Begin This Course

Before you begin this course, you should have thorough knowledge of SQL and knowledge of any procedural programming language. Required prerequisite is *Oracle Database 10g: SQL Fundamentals*.

How This Course Is Organized

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

Related Publications

Additional Publications

- System release bulletins
- Installation and user’s guides
- `read.me` files
- International Oracle User’s Group (IOUG) articles
- *Oracle Magazine*

Typographic Conventions

Following are two lists of typographical conventions that are used specifically within text or within code.

Typographic Conventions Within Text

Convention	Object or Term	Example
Uppercase	Commands, functions, column names, PL/SQL objects, schemas	Use the SELECT command to view information stored in the LAST_NAME column of the `employees` table.
Lowercase,	Filenames, variables, table names, usernames, passwords	where: *role* is the name of the role italic
Initial cap	Trigger and button names	Assign a When-Validate-Item trigger to the ORD block. Choose Cancel.
Italic	Books, names of courses and manuals, and emphasized words or phrases	For more information on the subject, see *Oracle9i Server SQL Language Reference Manual.* Do *not* save changes to the database.
Quotation marks	Lesson module titles referenced within a course	This subject is covered in Lesson 3, “Working with Objects.”

Typographic Conventions (continued)

Typographic Conventions Within Code

Convention	Object or Term	Example
Uppercase	Commands, functions	`SELECT userid` `FROM emp;`
Lowercase,	Syntax variables	`CREATE ROLE` *`role;`*

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

Lowercase	Column names, table names, filenames,
	
	PL/SQL objects
		`. . .`
		`OG_ACTIVATE_LAYER (OG_GET_LAYER ('prod_pie_layer'))`
		`. . .`
		
		`SELECT last_name`
		`FROM emp;`

Bold	Text that must be entered by a user	`DROP USER scott;`

Introduction

Objectives

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

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

ORACLE

Lesson Aim

This lesson gives you a high-level overview of the course and its flow. You learn about the database schema and the tables that the course uses. You are also introduced to different products in the Oracle 10*g* grid infrastructure.

Course Objectives

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

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

Course Objectives

This course presents the basics of PL/SQL. You learn about PL/SQL syntax, blocks, and programming constructs and about the advantages of integrating SQL with those constructs. You learn how to write PL/SQL program units and execute them efficiently. In addition, you learn how to use *i*SQL*Plus as a development environment for PL/SQL. You also learn how to design reusable program units, such as procedures and functions.

Course Agenda

Lessons for the first day:

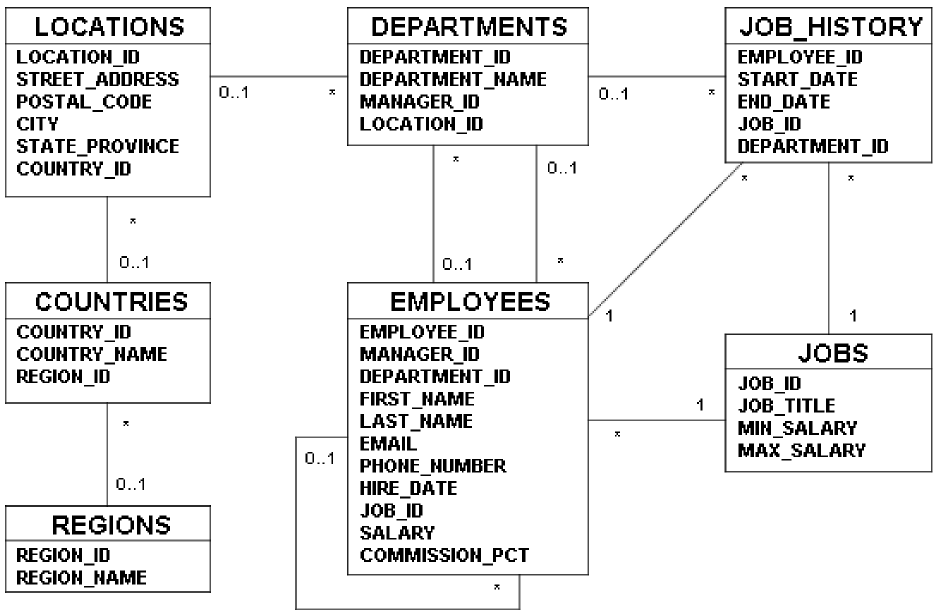
I. Introduction
1. Introduction to PL/SQL
2. Declaring PL/SQL Variables
3. Creating the Executable Section
4. Interacting with the Oracle Database Server
5. Writing Control Structures

Course Agenda

Lessons for the second day:

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

Human Resources (hr) Data Set



Human Resources (hr) Data Set

The Human Resources (hr) schema is a part of the Oracle sample schema that can be installed into an Oracle database. As the name indicates, the hr schema has tables that store all the information about all employees working in the organization. To reduce the complexity and volume of data, information about employees is stored in more than one table. For example, if an employee works in the education department, it is not necessary to store information about that employee as well as the education department in one table. Instead, you can store employee information in the employees table and department information in the departments table. This is how the hr schema is built.

The slide shows the hr schema tables and their relationships.

Table Descriptions

employees contains details about each employee working for a department. Some employees may not be assigned to a department.

departments contains details about the departments in which employees work. Each department may have a relationship representing the department manager in the employees table.

Human Resources (`hr`) Data Set (continued)

Table Descriptions (continued)

`jobs` contains the job types that can be held by each employee.

`job_history` contains the job histories of employees. If an employee changes departments within the job or changes jobs within the department, a new row is inserted into this table with the old job information of that employee.

`locations` contains the specific addresses of the offices, warehouses, and/or production sites of a company in a particular country.

`regions` contains rows representing a region (such as Americas, Asia, and so on).

`countries` contains rows for countries, each of which are associated with a region.

Note: This lesson introduces you to the various tables in the `hr` schema. If you want to see the data stored in each table, refer to Appendix B ("Table Descriptions and Data").

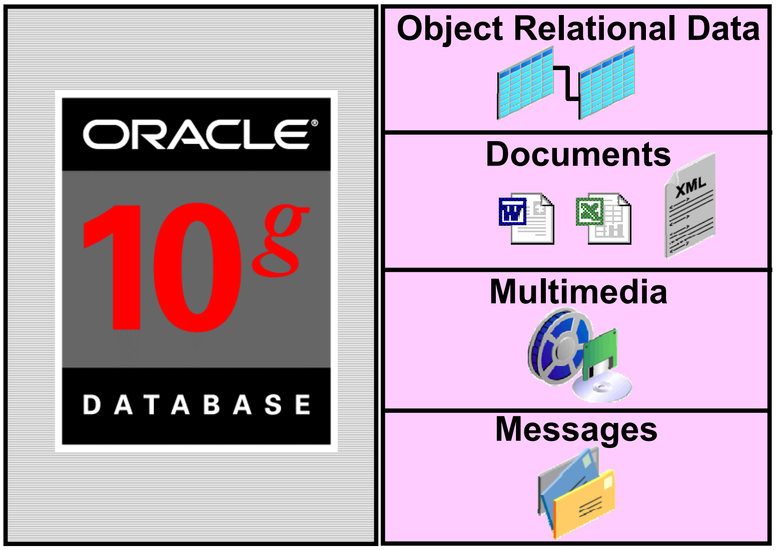
Oracle10*g* Grid Infrastructure

Oracle10*g* Grid Infrastructure

There are three grid infrastructure products in the Oracle10*g* release:

- Oracle Database 10*g*
- Oracle Application Server 10*g*
- Oracle Enterprise Manager 10*g* Grid Control

Oracle Database 10*g*



Oracle Database 10*g*

Oracle Database 10*g* is designed to store and manage enterprise information. By using Oracle Database 10*g*, management can reduce costs and be assured of a high quality of service. Reduced configuration and management requirements and automatic SQL tuning have significantly reduced the cost of maintaining the environment.

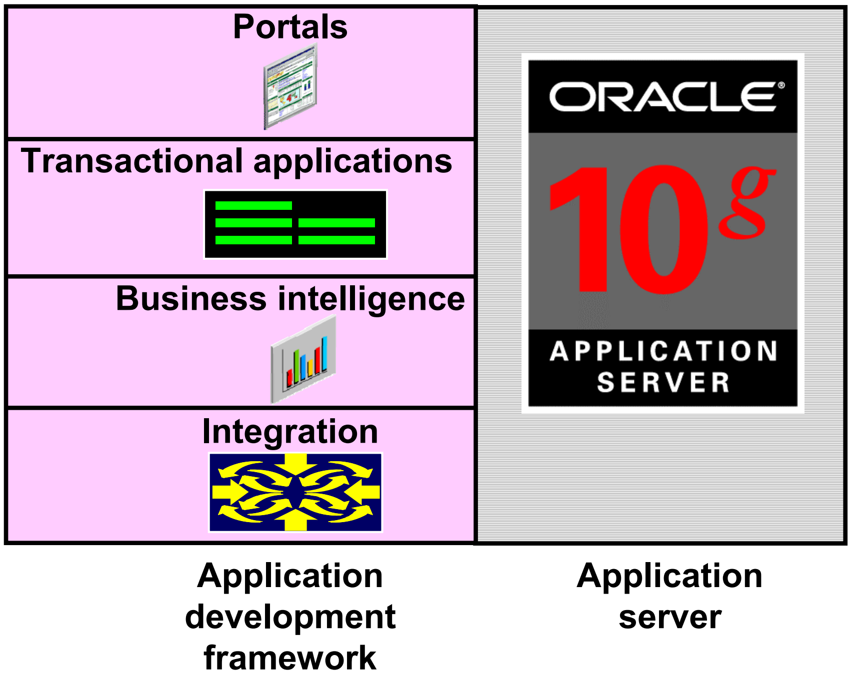
Oracle Database 10*g* contributes to the grid infrastructure products of the Oracle 10*g* release. Grid computing is all about computing as a utility. If you are a client, you need not know where your data resides or which computer stores it. You should be able to request information or do computations on your data and have it delivered to you.

Oracle Database 10*g* manages all your data. This is not just the object relational data that you expect an enterprise database to manage. It can also be unstructured data, such as:

- Spreadsheets
- Word documents
- PowerPoint presentations
- XML
- Multimedia data types (MP3, graphics, video, and so on)

The data does not even have to be in the database. Oracle Database 10*g* has services through which you can store metadata about information stored in file systems. You can use the database server to manage and serve information wherever it is located.

Oracle Application Server 10*g*



Oracle Application Server 10*g*

Oracle Application Server 10*g* provides a complete infrastructure platform for developing and deploying enterprise applications, integrating many functions including a J2EE and Web services run-time environment, an enterprise portal, an enterprise integration broker, business intelligence, Web caching, and identity management services. Oracle Application Server 10*g* adds new grid computing features, building on the success of Oracle9*i* Application Server, which has hundreds of customers running production enterprise applications.

Oracle Application Server 10*g* is the only application server to include services for all the different server applications that you want to run. It can run:

- Portals and Web sites
- Java transactional applications
- Business intelligence applications

It also provides integration between users, applications, and data throughout your organization.

Oracle Enterprise Manager 10*g* Grid Control

- **Software provisioning**
- **Application service-level monitoring**

Oracle Enterprise Manager 10*g* Grid Control

Oracle Enterprise Manager 10*g* Grid Control is a complete, integrated, central management console and underlying framework that automates administrative tasks across sets of systems in a grid environment. With Grid Control, you can group multiple hardware nodes, databases, application servers, and other targets into single logical entities. By executing jobs, enforcing standard policies, diagnosing and monitoring performance, and automating many other tasks across a group of targets instead of on many systems individually, Grid Control enables scaling with a growing grid.

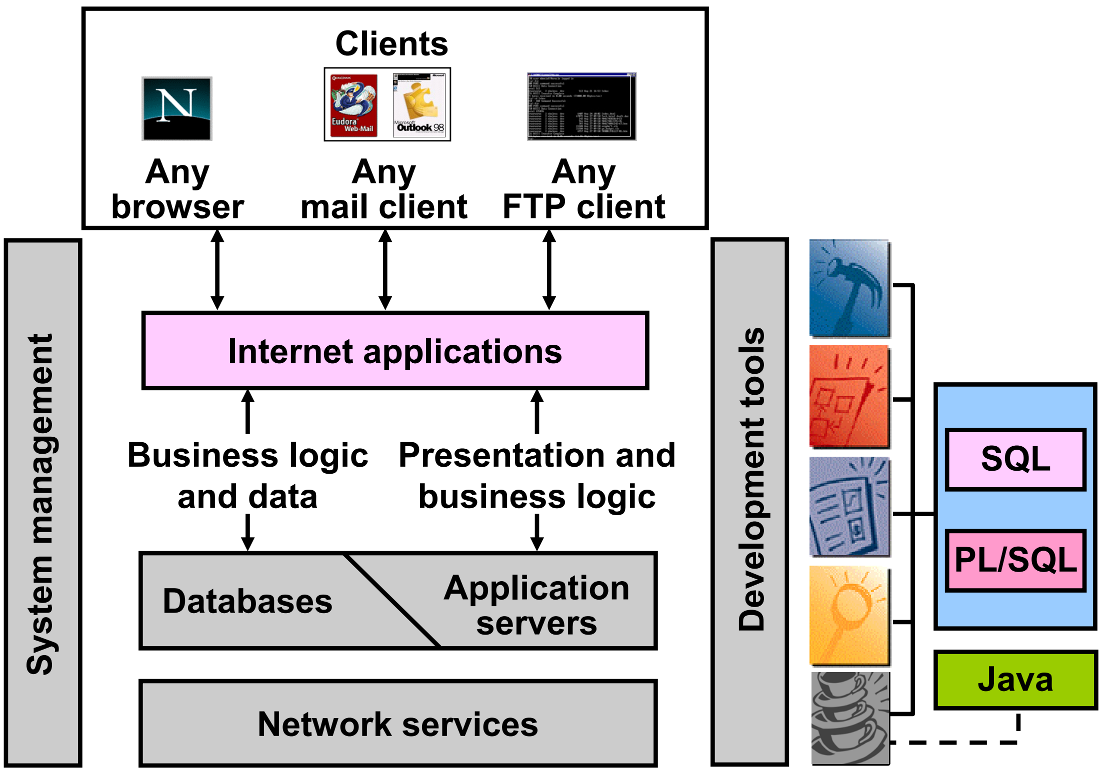
Software Provisioning

With Grid Control, the Oracle 10*g* platform automates installation, configuration, and cloning of Oracle Application Server 10*g* and Oracle Database 10*g* across multiples nodes. Oracle Enterprise Manager provides a common framework for software provisioning and management, allowing administrators to create, configure, deploy, and utilize new servers with new instances of the application server and database as they are needed.

Application Service-Level Monitoring

Grid Control views the availability and performance of the grid infrastructure as a unified whole, as a user would experience it, rather than as isolated storage units, processing boxes, databases, and application servers.

Oracle Internet Platform



Oracle Internet Platform

To develop an e-commerce application, you need a product that can store and manage the data, a product that can provide a run-time environment for your applications implementing business logic, and a product that can monitor and diagnose the application after it is integrated. Oracle 10*g* grid infrastructure products, discussed earlier, provide all the necessary components to develop your enterprise. Oracle offers a comprehensive, high-performance Internet platform for e-commerce and data warehousing. This integrated platform includes everything needed to develop, deploy, and manage Internet applications.

The Oracle Internet Platform is built on three core pieces:

- Browser-based clients to process presentation
- Application servers to execute business logic and serve presentation logic to browser-based clients
- Databases to execute database-intensive business logic and server data

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

Summary

In this lesson, you should have learned how to:

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

Course Practices

When you perform the practices in the course, you develop a simple application using an anonymous block. This anonymous block covers the following:

- Writing a declarative section
- Declaring variables of scalar types
- Declaring variables using the `%TYPE` attribute
- Writing an executable section
- Accepting user inputs for variables
- Retrieving the values from the database and storing the values in the variables by using the `INTO` clause
- Writing a nested block within the executable section
- Using the control structures in the executable section to perform business logic
- Using the `INDEX BY` table to store values and print them
- Handling exceptions

What Is the Functionality of This Application?

This application is a simple HR application, and only employees working in the Human Resources department are authorized to use it. In the `employees` table, only one employee is in the HR department. Therefore, you can use `employee_id` for authentication.

The company has decided to provide salary raises to employees in certain departments this quarter. The raise percentages are determined by the employees' current salaries.

Employees in the following departments are eligible for raises this quarter:

department_id	**department_name**
20	Marketing
60	IT
80	Sales
100	Finance
110	Accounting

The salary ranges and the resulting raise percentages are as follows:

salary	**Raise percentage**
< 6500	20
> 6500 < 9500	15
> 9500 <12000	8
>12000	3

Introduction to PL/SQL

Objectives

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

- **Explain the need for PL/SQL**
- **Explain the benefits of PL/SQL**
- **Identify the different types of PL/SQL blocks**
- **Use *i*SQL*Plus as a development environment for PL/SQL**
- **Output messages in PL/SQL**

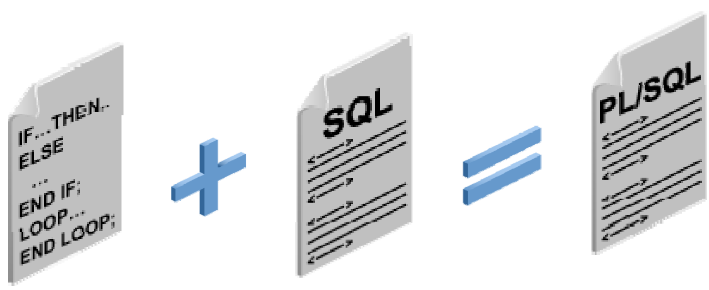
Lesson Aim

This lesson introduces PL/SQL and PL/SQL programming constructs. You learn about the benefits of PL/SQL. You also learn to use *i*SQL*Plus as a development environment for PL/SQL.

What Is PL/SQL?

PL/SQL:

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



What Is PL/SQL?

Structured Query Language (SQL) is the primary language used to access and modify data in relational databases. There are only a few SQL commands, so you can easily learn and use them. Consider an example:

```
SELECT first_name, department_id, salary FROM employees;
```

The SQL statement shown above is simple and straightforward. However, if you want to alter any data that is retrieved in a conditional manner, you soon encounter the limitations of SQL.

Consider a slightly modified problem statement: For every employee retrieved, check the `department_id` and the salary. Depending on the department's performance and also the employee's `salary`, you may want to provide varying bonuses to the employees.

Looking at the problem, you know that you have to execute the preceding SQL statement, collect the data, and apply logic to the data. One solution is to write a SQL statement for each department to give bonuses to the employees in that department. Remember that you also have to check the salary component before deciding the bonus amount. This makes it a little complicated. You now feel that it would be much easier if you had conditional statements. PL/SQL is designed to meet such requirements. It provides a programming extension to already-existing SQL.

About PL/SQL

PL/SQL:

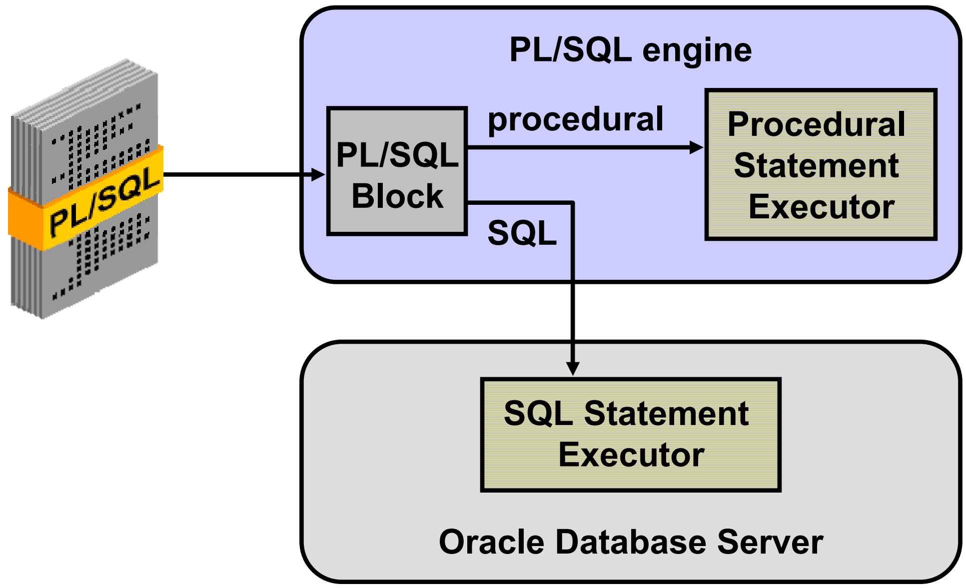
- Provides a block structure for executable units of code. Maintenance of code is made easier with such a well-defined structure.
- Provides procedural constructs such as:
  – Variables, constants, and types
  – Control structures such as conditional statements and loops
  – Reusable program units that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

About PL/SQL

PL/SQL defines a block structure for writing code. Maintaining and debugging the code is made easier with such a structure. One can easily understand the flow and execution of the program unit.

PL/SQL offers modern software engineering features such as data encapsulation, exception handling, information hiding, and object orientation. It brings state-of-the-art programming to the Oracle server and toolset. PL/SQL provides all the procedural constructs that are available in any third-generation language (3GL).

PL/SQL Environment



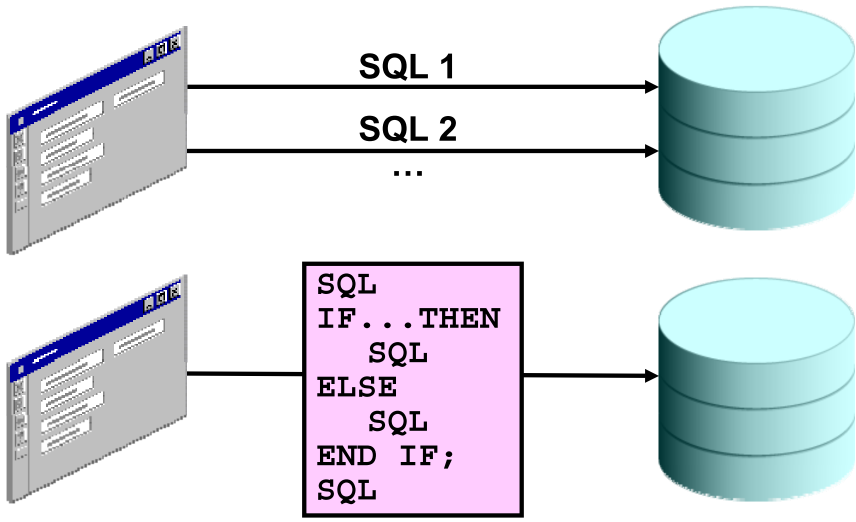
PL/SQL Environment

The slide shows the PL/SQL execution environment in the Oracle database server. A PL/SQL block contains procedural statements and SQL statements. When you submit the PL/SQL block to the server, the PL/SQL engine first parses the block. The PL/SQL engine identifies the procedural statements and SQL statements. It passes the procedural statements to the procedural statement executor and passes the SQL statements to the SQL statement executor individually.

The diagram in the slide shows the PL/SQL engine within the database server. The Oracle application development tools can also contain a PL/SQL engine. The tool passes the blocks to its local PL/SQL engine. Therefore, all procedural statements are executed locally and only the SQL statements are executed in the database. The engine used depends on where the PL/SQL block is being invoked from.

Benefits of PL/SQL

- Integration of procedural constructs with SQL
- Improved performance



Benefits of PL/SQL

Integration of procedural constructs with SQL: The most important advantage of PL/SQL is the integration of procedural constructs with SQL. SQL is a nonprocedural language. When you issue a SQL command, your command tells the database server *what* to do. However, you cannot specify *how* to do it. PL/SQL integrates control statements and conditional statements with SQL, giving you better control of your SQL statements and their execution. Earlier in this lesson, you saw an example of the need for such integration.

Improved performance: Without PL/SQL, you would not be able to logically combine SQL statements as one unit. If you have designed an application containing forms, you may have many different forms with fields in each form. When a form submits the data, you may have to execute a number of SQL statements. SQL statements are sent to the database one at a time. This results in many network trips and one call to the database for each SQL statement, thereby increasing network traffic and reducing performance (especially in a client/server model).

With PL/SQL, you can combine all these SQL statements into a single program unit. The application can send the entire block to the database instead of sending the SQL statements one at a time. This significantly reduces the number of database calls. As the slide illustrates, if the application is SQL intensive, you can use PL/SQL blocks to group SQL statements before sending them to the Oracle database server for execution.

Benefits of PL/SQL

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

Benefits of PL/SQL (continued)

Modularized program development: A basic unit in all PL/SQL programs is the block. Blocks can be in a sequence or they can be nested in other blocks. Modularized program development has the following advantages:

- You can group logically related statements within blocks.
- You can nest blocks inside larger blocks to build powerful programs.
- You can break your application into smaller modules. If you are designing a complex application, PL/SQL allows you to break down the application into smaller, manageable, and logically related modules.
- You can easily maintain and debug the code.

Integration with tools: The PL/SQL engine is integrated in Oracle tools such as Oracle Forms, Oracle Reports, and so on. When you use these tools, the locally available PL/SQL engine processes the procedural statements; only the SQL statements are passed to the database.

Benefits of PL/SQL (continued)

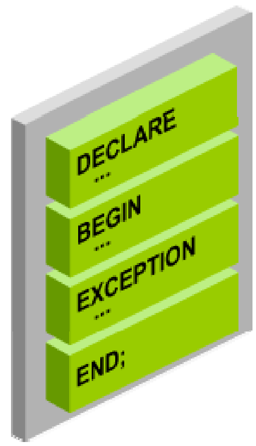
Portability: PL/SQL programs can run anywhere an Oracle server runs, irrespective of the operating system and the platform. You do not need to tailor them to each new environment. You can write portable program packages and create libraries that can be reused in different environments.

Exception handling: PL/SQL enables you to handle exceptions efficiently. You can define separate blocks for dealing with exceptions. You will learn more about exception handling later in the course.

PL/SQL shares the same data type system as SQL (with some extensions) and uses the same expression syntax.

PL/SQL Block Structure

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



PL/SQL Block Structure

The slide shows a basic PL/SQL block. A PL/SQL block consists of three sections:

- **Declarative (optional):** The declarative section begins with the keyword `DECLARE` and ends when the executable section starts.
- **Executable (required):** The executable section begins with the keyword `BEGIN` and ends with `END`. 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 exception section is nested within the executable section. This section begins with the keyword `EXCEPTION`.

PL/SQL Block Structure (continued)

In a PL/SQL block, the keywords `DECLARE`, `BEGIN`, and `EXCEPTION` are not terminated by a semicolon. However, the keyword `END`, all SQL statements, and PL/SQL statements must be terminated with a semicolon.

Section	Description	Inclusion
Declarative (`DECLARE`)	Contains declarations of all variables, constants, cursors, and user-defined exceptions that are referenced in the executable and exception sections	Optional
Executable (`BEGIN … END`)	Contains SQL statements to retrieve data from the database; contains PL/SQL statements to manipulate data in the block	Mandatory
Exception (`EXCEPTION`)	Specifies the actions to perform when errors and abnormal conditions arise in the executable section	Optional

Block Types

Anonymous	Procedure	Function

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

Block Types

A PL/SQL program comprises one or more blocks. These blocks can be entirely separate or nested within another block. There are three types of blocks that make up a PL/SQL program. They are:

- Anonymous blocks
- Procedures
- Functions

Anonymous blocks: Anonymous blocks are unnamed blocks. They are declared inline at the point in an application where they are to be executed and are compiled each time the application is executed. These blocks are not stored in the database. They are passed to the PL/SQL engine for execution at run time. Triggers in Oracle Developer components consist of such blocks. These anonymous blocks get executed at run time because they are inline. If you want to execute the same block again, you have to rewrite the block. You are unable to invoke or call the block that you wrote earlier because blocks are anonymous and do not exist after they are executed.

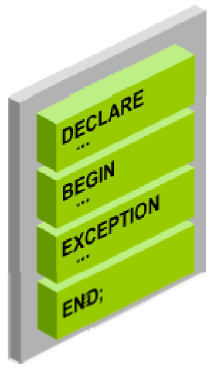
Block Types (continued)

Subprograms: Subprograms are complementary to anonymous blocks. They are named PL/SQL blocks that are stored in the database. Because they are named and stored, you can invoke them whenever you want (depending on your application). You can declare them either as procedures or as functions. You typically use a procedure to perform an action and a function to compute and return a value.

You can store subprograms at the server or application level. Using Oracle Developer components (Forms, Reports), you can declare procedures and functions as part of the application (a form or report) and call them from other procedures, functions, and triggers within the same application whenever necessary.

Note: A function is similar to a procedure, except that a function must return a value.

Program Constructs



Tools Constructs	Database Server Constructs
Anonymous blocks	Anonymous blocks
Application procedures or functions	Stored procedures or functions
Application packages	Stored packages
Application triggers	Database triggers
Object types	Object types

Program Constructs

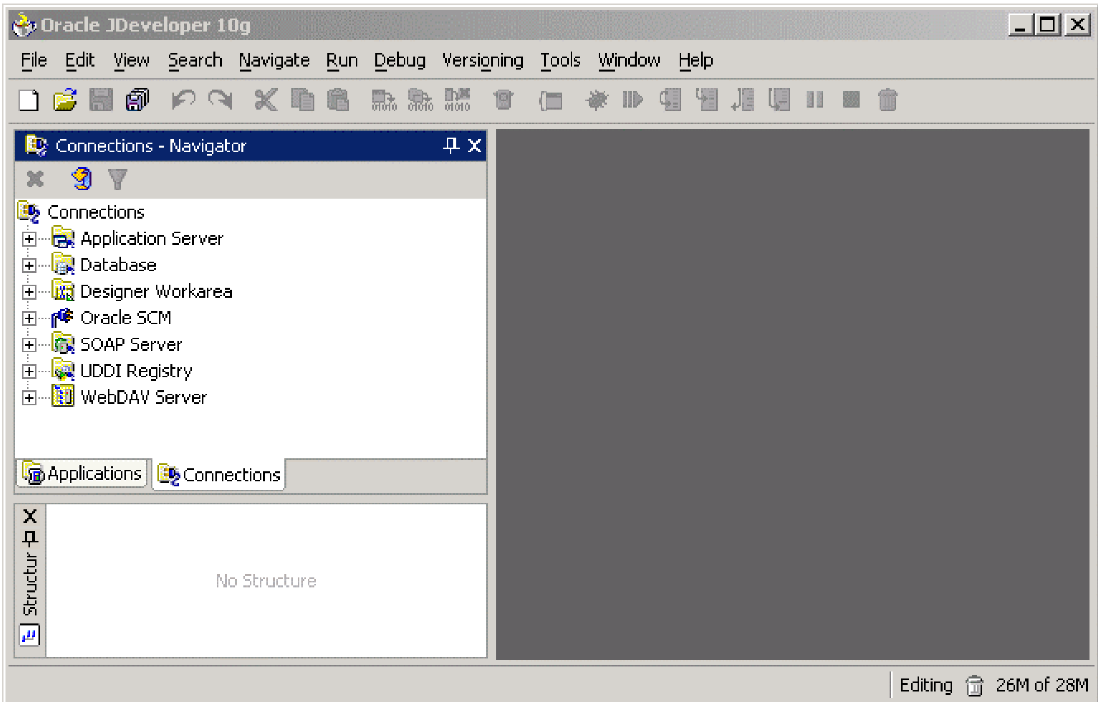
The following table outlines a variety of different PL/SQL program constructs that use the basic PL/SQL block. The program constructs are available based on the environment in which they are executed.

Program Construct	Description	Availability
Anonymous blocks	Unnamed PL/SQL blocks that are embedded within an application or are issued interactively	All PL/SQL environments
Application procedures or functions	Named PL/SQL blocks stored in an Oracle Forms Developer application or shared library; can accept parameters and can be invoked repeatedly by name	Oracle Developer tools components (for example, Oracle Forms Developer, Oracle Reports)
Stored procedures or functions	Named PL/SQL blocks stored in the Oracle server; can accept parameters and can be invoked repeatedly by name	Oracle server or Oracle Developer tools
Packages (application or stored)	Named PL/SQL modules that group related procedures, functions, and identifiers	Oracle server and Oracle Developer tools components (for example, Oracle Forms Developer)

Program Constructs (continued)

Program Construct	Description	Availability
Database triggers	PL/SQL blocks that are associated with a database table and fired automatically when triggered by various events	Oracle server or any Oracle tool that issues the DML
Application triggers	PL/SQL blocks that are associated either with a database table or system events. They are fired automatically when triggered by a DML or a system event respectively.	Oracle Developer tools components (for example, Oracle Forms Developer)
Object types	User-defined composite data types that encapsulate a data structure along with the functions and procedures needed to manipulate the data	Oracle server and Oracle Developer tools

PL/SQL Programming Environments



PL/SQL Programming Environments

Oracle JDeveloper 10*g*: An integrated development environment (IDE) that provides end-to-end support for building, testing, and deploying J2EE applications, Web services, and PL/SQL

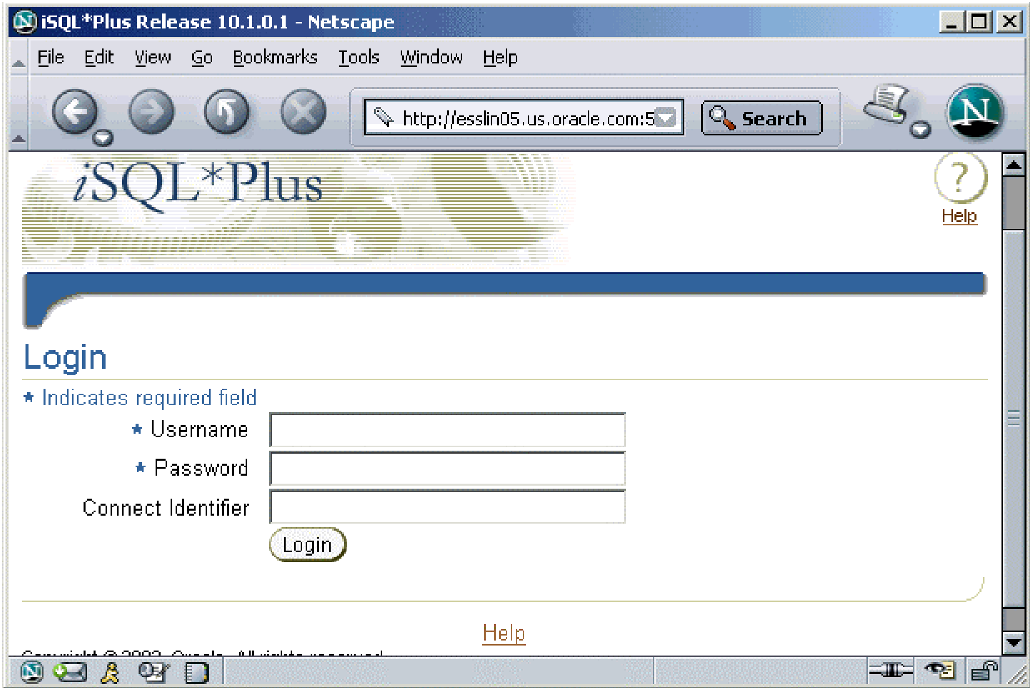
You can use Oracle JDeveloper 10*g* to do the following:

- Establish connection to the database with a user-friendly wizard
- Browse through the objects in the database you are connected to
- Create database users and objects
- Create, run, and debug PL/SQL programs such as procedures, functions, and packages

Note: Oracle JDeveloper 10*g* and *i*SQL*Plus can both be used as programming environments. However, this course uses *i*SQL*Plus for all demonstrations and practices.

PL/SQL Programming Environments

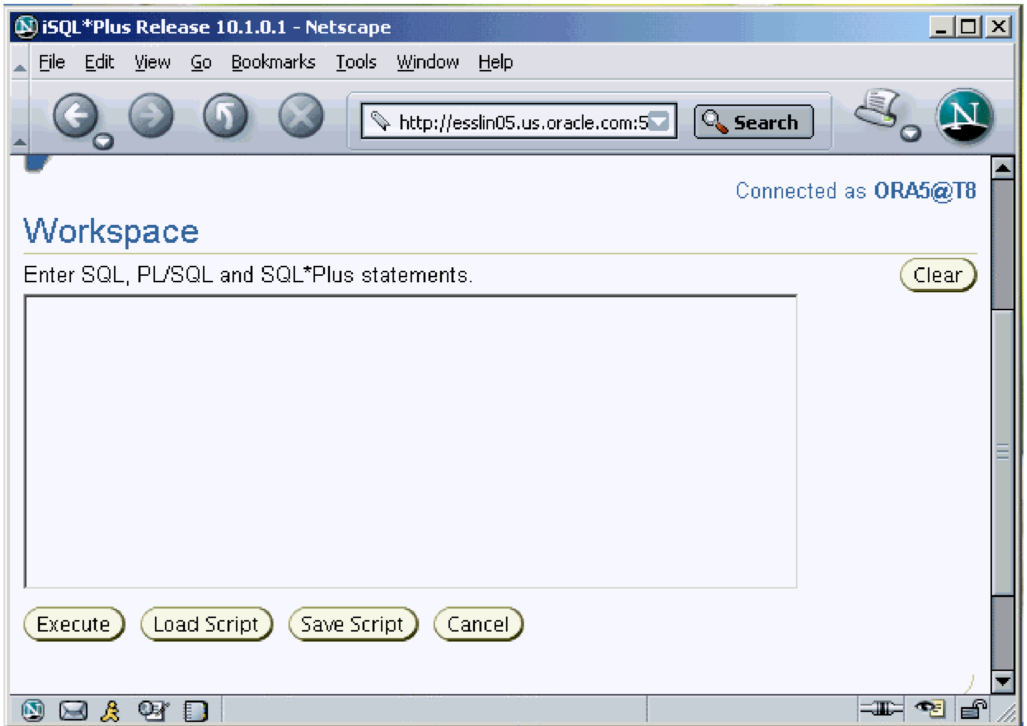
*i*SQL*Plus



PL/SQL Programming Environments (continued)

***i*SQL*Plus:** A browser-based interface to SQL*Plus. You can connect to the local database or remote database by using *i*SQL*Plus. It enables you to perform all the operations that you can perform with the command-line version of SQL*Plus.

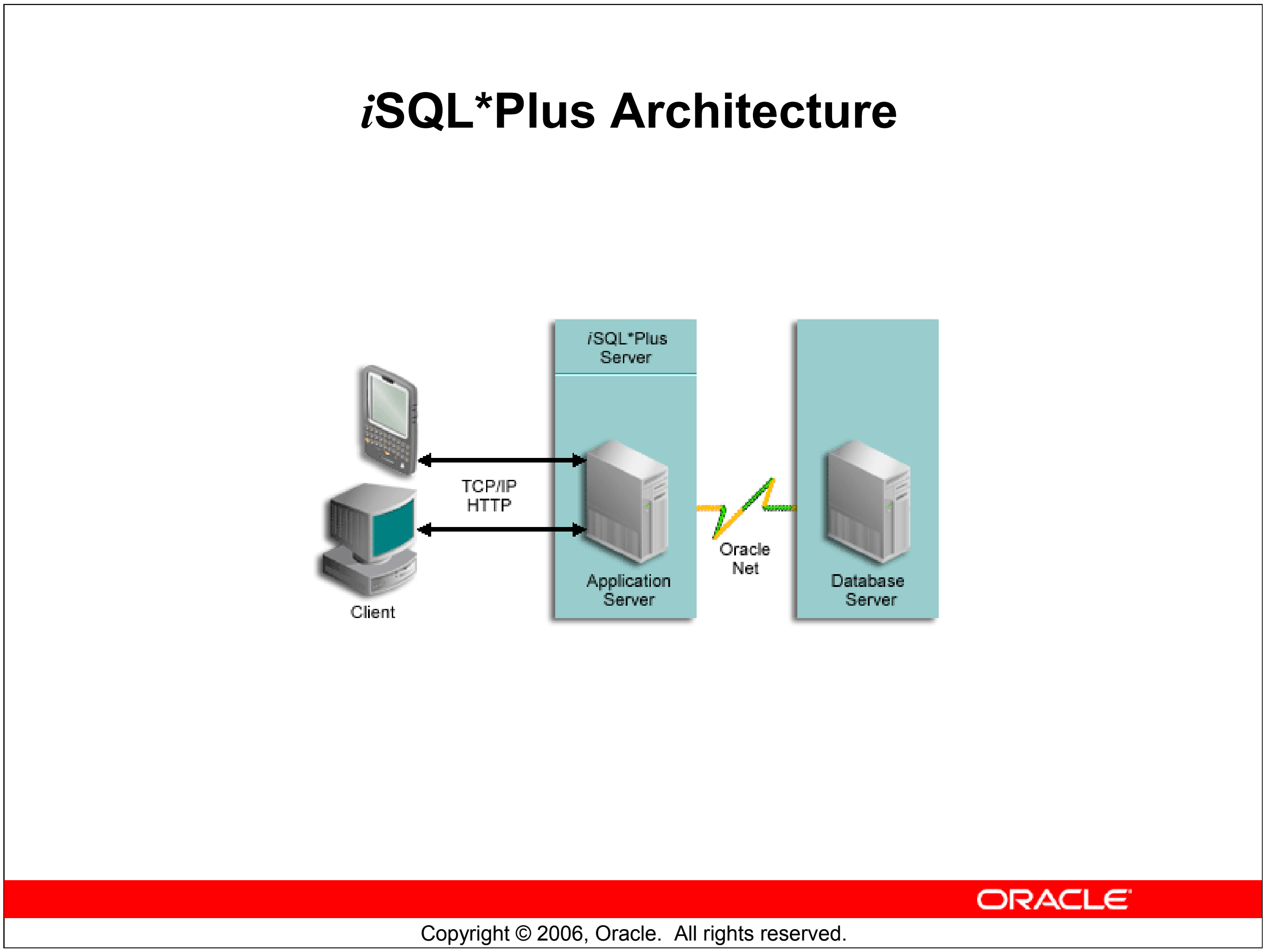
PL/SQL Programming Environments



ORACLE

PL/SQL Programming Environments (continued)

When you log in to *i*SQL*Plus, you see the screen shown in the slide. Note that you have a workspace to enter SQL, PL/SQL, and SQL*Plus statements. Click the Execute button to execute your statements in the workspace. Click the Save Script button when you want to save all the commands in the workspace in a script file. You can save the script as a `*.sql` file. If you want to execute any script file, click the Load Script button and browse to select the script file. All the statements in the script file are loaded to the workspace and you can click the Execute button to execute the statements. The Clear button is used to clear the workspace.



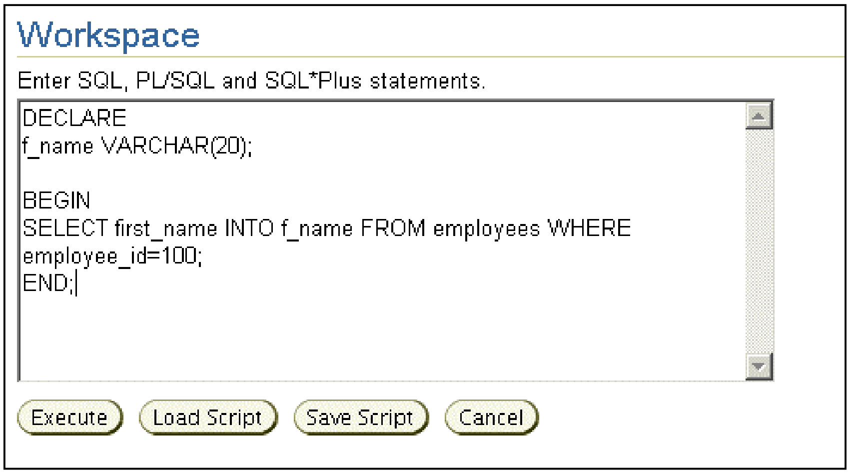
*i*SQL*Plus Architecture

*i*SQL*Plus uses a three-tier model as shown in the slide. The three tiers in the architecture are:

- **Client tier:** The client is a typical HTTP client. Any browser connected to the intranet or Internet can access the *i*SQL*Plus user interface.
- **Middle tier:** The application server forms the middle tier in the *i*SQL*Plus architecture. The application server is installed when the database is installed. The *i*SQL*Plus server must be installed on the same machine as the application server. The middle tier is a Java2 Enterprise Edition (J2EE)–compliant application server. The application server enables communication between *i*SQL*Plus and the database. The three tiers in the architecture need not be on the same machine. However, the HTTP Server and *i*SQL*Plus Server should be on the same machine. *i*SQL*Plus manages a unique identity for each session. The advantage of this is that many concurrent users can use *i*SQL*Plus to access the database.
- **Database tier:** The database tier has the database server. The Oracle Net components enable communication between the *i*SQL*Plus Server and the database.

Create an Anonymous Block

Type the anonymous block in the *i*SQL*Plus workspace:

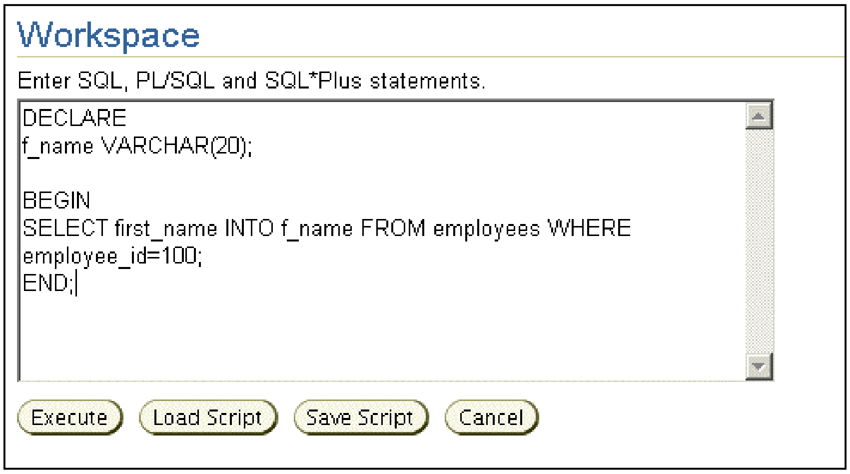


Create an Anonymous Block

To create an anonymous block using *i*SQL*Plus, enter the block in the workspace (as shown in the slide). The block has the declarative section and the executable section. You need not pay attention to the syntax of statements in the block; you learn the syntax later in the course. The anonymous block gets the `first_name` of the employee whose `employee_id` is 100 and stores it in a variable called `f_name`.

Execute an Anonymous Block

Click the Execute button to execute the anonymous block:



PL\SQL procedure successfully completed.

Execute an Anonymous Block

Click the Execute button to execute the anonymous block in the workspace. Note that the message "PL\SQL procedure successfully completed" is displayed after the block is executed.

Test the Output of a PL/SQL Block

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

```
SET SERVEROUTPUT ON
…
DBMS_OUTPUT.PUT_LINE(' The First Name of the
Employee is ' || f_name);
…
```

ORACLE

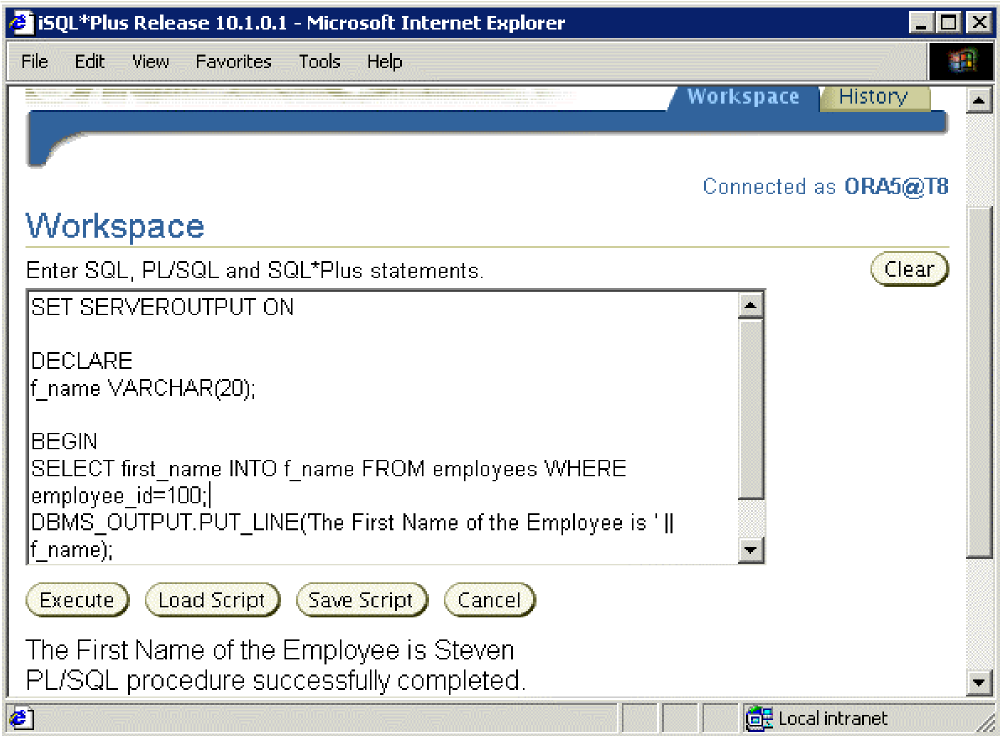
Test the Output of a PL/SQL Block

In the example shown in the previous slide, we have stored a value in the variable `f_name`. However, we have not printed the value. You now learn how to print the value.

PL/SQL does not have built-in input or output functionality. Therefore, we use predefined Oracle packages for input and output. To generate output, you must:

- Enable output in *i*SQL*Plus by using the `SET SERVEROUTPUT ON` command. `SET SERVEROUTPUT ON` is a SQL*Plus command that is also supported by *i*SQL*Plus.
- Use the procedure `PUT_LINE` of the package `DBMS_OUTPUT` to display the output. Pass the value that has to be printed as argument to this procedure (as shown in the slide). The procedure then outputs the arguments.

Test the Output of a PL/SQL Block



Test the Output of a PL/SQL Block (continued)

The slide shows the output of the PL/SQL block after the inclusion of the code for generating output.

Summary

In this lesson, you should have learned how to:

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

Summary

PL/SQL is a language that has programming features that serve as an extension to SQL. SQL, which is a nonprocedural language, is made procedural with PL/SQL programming constructs. PL/SQL applications can run on any platform or operating system on which an Oracle server runs. In this lesson, you learned how to build basic PL/SQL blocks.

Practice 1: Overview

This practice covers the following topics:

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

Practice 1: Overview

This practice reinforces the basics of PL/SQL covered in this lesson.

- Exercise 1 is a paper-based exercise in which you identify PL/SQL blocks that execute successfully.
- Exercise 2 involves creating and executing a simple PL/SQL block.

Practice 1

Before you begin this practice, please ensure that you have seen both the viewlets on *i*SQL*Plus usage.

The `labs` folder will be your working directory. You can save your scripts in the `labs` folder. Please take the instructor's help to locate the `labs` folder for this course. The solutions for all practices are in the `soln` folder.

1. Which of the following PL/SQL blocks execute successfully?

   a.
   ```
   BEGIN
   END;
   ```

   b.
   ```
   DECLARE
   amount  INTEGER(10);
   END;
   ```

   c.
   ```
   DECLARE
   BEGIN
   END;
   ```

   d.
   ```
   DECLARE
   amount INTEGER(10);
   BEGIN
   DBMS_OUTPUT.PUT_LINE(amount);
   END;
   ```

2. Create and execute a simple anonymous block that outputs "Hello World." Execute and save this script as `lab_01_02_soln.sql`.

Declaring PL/SQL Variables

Objectives

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

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

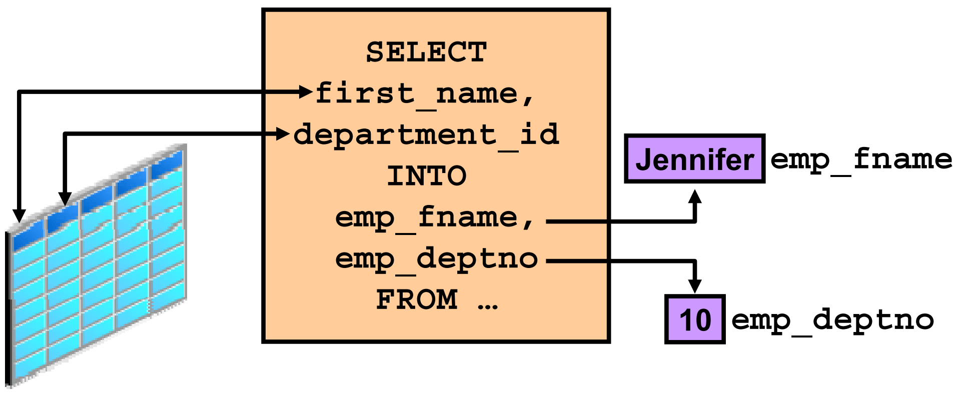
Lesson Aim

You have already learned about basic PL/SQL blocks and their sections. In this lesson, you learn about valid and invalid identifiers. You learn how to declare and initialize variables in the declarative section of a PL/SQL block. The lesson describes the various data types. You also learn about the `%TYPE` attribute and its benefits.

Use of Variables

Variables can be used for:

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



Use of Variables

With PL/SQL you can declare variables and then use them in SQL and procedural statements.

Variables are mainly used for storage of data and manipulation of stored values. Consider the SQL statement shown in the slide. The statement retrieves the `first_name` and `department_id` from the table. If you have to manipulate the `first_name` or the `department_id`, then you have to store the retrieved value. Variables are used to temporarily store the value. You can use the value stored in these variables for processing and manipulating the data. Variables can store any PL/SQL object, such as variables, types, cursors, and subprograms.

Reusability is another advantage of declaring variables. After they are declared, variables can be used repeatedly in an application by referring to them in the statements.

Identifiers

Identifiers are used for:

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

Identifiers

Identifiers are mainly used to provide conventions for naming variables. The rules for naming a variable are listed in the slide.

What Is the Difference Between a Variable and an Identifier?

Identifiers are names of variables. Variables are storage locations of data. Data is stored in memory. Variables point to this memory location where data can be read and modified. Identifiers are used to *name* PL/SQL objects (such as variables, types, cursors, and subprograms). Variables are used to *store* PL/SQL objects.

Handling Variables in PL/SQL

Variables are:

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

Handling Variables in PL/SQL

Declared and Initialized in the Declaration Section

You can declare variables in the declarative part of any PL/SQL block, subprogram, or package. Declarations allocate storage space for a value, specify its data type, and name the storage location so that you can reference it. Declarations can also assign an initial value and impose the `NOT NULL` constraint on the variable. Forward references are not allowed. You must declare a variable before referencing it in other statements, including other declarative statements.

Used and Assigned New Values in the Executable Section

In the executable section, the existing value of the variable can be replaced with the new value.

Passed as Parameters to PL/SQL Subprograms

Subprograms can take parameters. You can pass variables as parameters to subprograms.

Used to Hold the Output of a PL/SQL Subprogram

You have learned that the only difference between procedures and functions is that functions must return a value. Variables can be used to hold the value that is returned by a function.

Declaring and Initializing PL/SQL Variables

Syntax

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

Examples

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


Declaring and Initializing PL/SQL Variables

You must declare all PL/SQL identifiers in the declaration section before referencing them in the PL/SQL block. You have the option of assigning an initial value to a variable (as shown in the slide). You do not need to assign a value to a variable in order to declare it. If you refer to other variables in a declaration, you must be sure to declare them separately in a previous statement.

In the syntax:

identifier	Is the name of the variable
CONSTANT	Constrains the variable so that its value cannot change (Constants must be initialized.)
data type	Is a scalar, composite, reference, or LOB data type (This course covers only scalar, composite, and LOB data types.)
NOT NULL	Constrains the variable so that it must contain a value (NOT NULL variables must be initialized.)
expr	Is any PL/SQL expression that can be a literal expression, another variable, or an expression involving operators and functions

Note: In addition to variables, you can also declare cursors and exceptions in the declarative section. You learn how to declare cursors and exceptions later in the course.

Declaring and Initializing PL/SQL Variables

1

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

2

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

ORACLE

Declaring and Initializing PL/SQL Variables (continued)

Examine the two code blocks in the slide.

1. The variable `Myname` is declared in the declarative section of the block. This variable can be accessed in the executable section of the same block. A value `John` is assigned to the variable in the executable section. String literals must be enclosed in single quotation marks. If your string has a quotation mark as in " Today's Date", then the string would be " `Today's Date`". ':=' is the assignment operator. The procedure `PUT_LINE` is invoked by passing the variable `Myname`. The value of the variable is concatenated with the string 'My name is: '. The output of this anonymous block is:

```
My name is:
My name is: John
PL/SQL procedure successfully completed.
```

2. In the second block, the variable `Myname` is declared and initialized in the declarative section. `Myname` holds the value `John` after initialization. This value is manipulated in the executable section of the block. The output of this anonymous block is:

```
My name is: Steven
PL/SQL procedure successfully completed.
```

Delimiters in String Literals

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

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


Delimiters in String Literals

If your string contains an apostrophe (identical to a single quotation mark), you must double the quotation mark, as in the following example:

```
event VARCHAR2(15):='Father''s day';
```

The first quotation mark acts as the escape character. This makes your string complicated, especially if you have SQL statements as strings. You can specify any character that is not present in the string as delimiter. The slide shows how to use the `q'` notation to specify the delimiter. The examples use '!' and '[' as delimiters. Consider the following example:

```
event := q'!Father's day!';
```

You can compare this with the first example on this notes page. You start the string with `q'` if you want to use a delimiter. The character following the notation is the delimiter used. Enter your string after specifying the delimiter, close the delimiter, and close the notation with a single quotation mark. The following example shows how to use '[' as a delimiter:

```
event := q'[Mother's day]';
```

Types of Variables

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

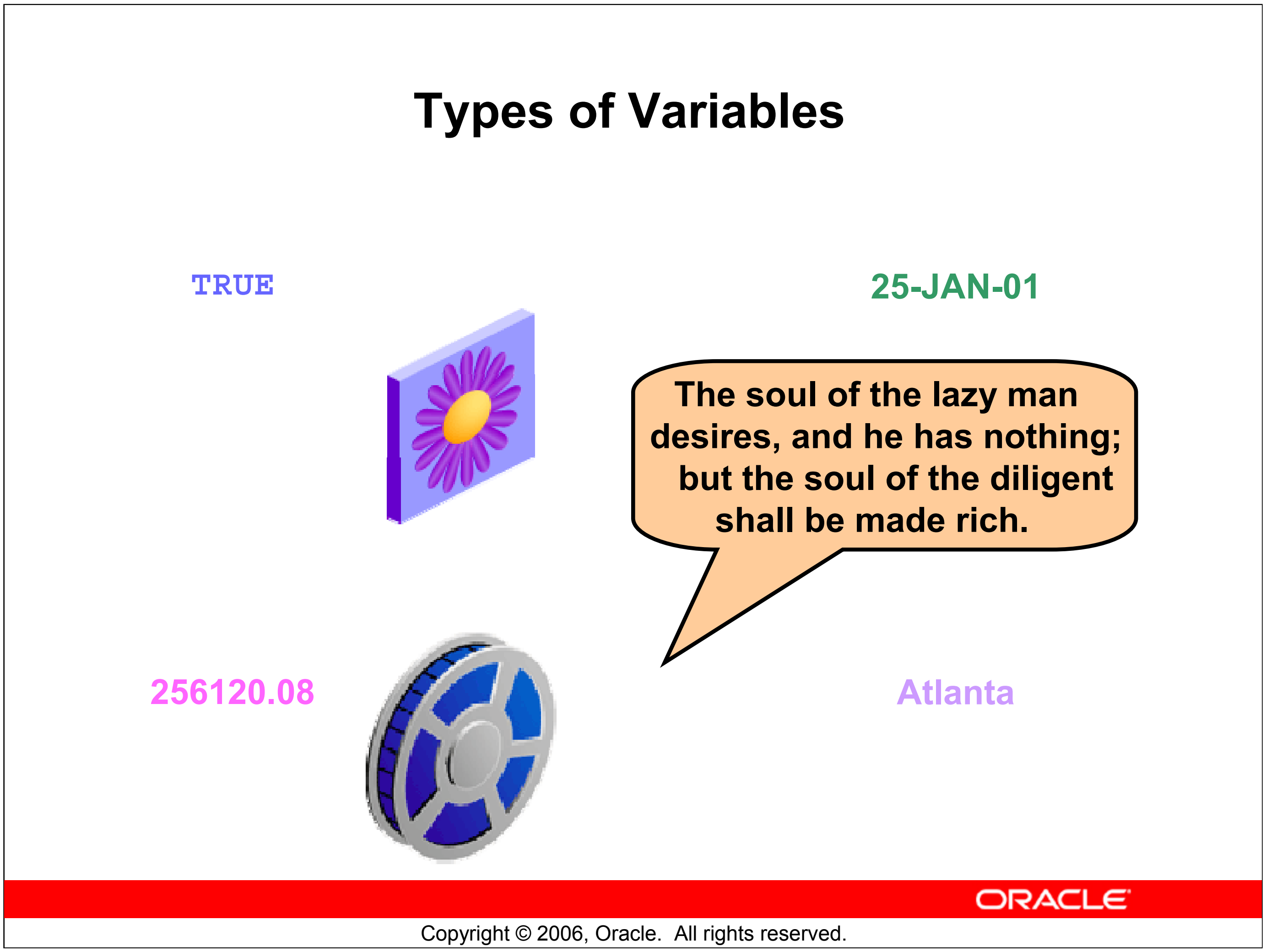
Types of Variables

All PL/SQL variables have a data type, which specifies a storage format, constraints, and a valid range of values. PL/SQL supports five data type categories—scalar, composite, reference, large object (LOB), and object—that you can use for declaring variables, constants, and pointers.

- **Scalar data types:** Scalar data types hold a single value. The value depends on the data type of the variable. For example, the variable `Myname` in the example in slide 7 is of type `VARCHAR2`. Therefore, `Myname` can hold a string value. PL/SQL also supports Boolean variables.
- **Composite data types:** Composite data types contain internal elements that are either scalar or composite. Record and table are examples of composite data types.
- **Reference data types:** Reference data types hold values, called *pointers*, that point to a storage location.
- **LOB data types:** LOB data types hold values, called *locators*, that specify the location of large objects (such as graphic images) that are stored out of line.

Non-PL/SQL variables include host language variables declared in precompiler programs, screen fields in Forms applications, and *i*SQL*Plus host variables. You learn about host variables later in this lesson.

For more information about LOBs, see the *PL/SQL User's Guide and Reference*.



Types of Variables (continued)

The slide illustrates the following data types:

- `TRUE` represents a Boolean value.
- 25-JAN-01 represents a `DATE`.
- The image represents a `BLOB`.
- The text of the proverb can represent a `VARCHAR2` data type or a `CLOB`.
- 256120.08 represents a `NUMBER` data type with precision and scale.
- The film reel represents a `BFILE`.
- The city name *Atlanta* represents a `VARCHAR2`.

Guidelines for Declaring and Initializing PL/SQL Variables

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

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

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

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

Guidelines for Declaring and Initializing PL/SQL Variables

Here are some guidelines to follow when you declare PL/SQL variables.

- Follow naming conventions: for example, `name` to represent a variable and `c_name` to represent a constant.
- Use meaningful and appropriate names for variables. For example, consider using `salary` and `sal_with_commission` instead of `salary1` and `salary2`.
- If you use the `NOT NULL` constraint, you must assign a value when you declare the variable.
- In constant declarations, the keyword `CONSTANT` must precede the type specifier. The following declaration names a constant of `NUMBER` subtype `REAL` and assigns the value of 50,000 to the constant. A constant must be initialized in its declaration; otherwise, you get a compilation error. After initializing a constant, you cannot change its value.

```
sal CONSTANT REAL := 50000.00;
```

Guidelines for Declaring PL/SQL Variables

- **Avoid using column names as identifiers.**

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

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

Guidelines for Declaring PL/SQL Variables

- Initialize the variable to an expression with the assignment operator (`:=`) or with the `DEFAULT` reserved word. If you do not assign an initial value, the new variable contains `NULL` by default until you assign a value. To assign or reassign a value to a variable, you write a PL/SQL assignment statement. It is good programming practice to initialize all variables.
- Two objects can have the same name only if they are defined in different blocks. Where they coexist, you can qualify them with labels and use them.
- Avoid using column names as identifiers. If PL/SQL variables occur in SQL statements and have the same name as a column, the Oracle server assumes that it is the column that is being referenced. Although the example code in the slide works, code that is written using the same name for a database table and variable name is not easy to read or maintain.
- Impose the `NOT NULL` constraint when the variable must contain a value. You cannot assign nulls to a variable defined as `NOT NULL`. The `NOT NULL` constraint must be followed by an initialization clause.

```
pincode NUMBER(15) NOT NULL := 'Oxford';
```

Scalar Data Types

- Hold a single value
- Have no internal components

TRUE

25-JAN-01

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

256120.08

Atlanta

Scalar Data Types

Every constant, variable, and parameter has a data type that specifies a storage format, constraints, and valid range of values. PL/SQL provides a variety of predefined data types. For instance, you can choose from integer, floating point, character, Boolean, date, collection, and LOB types. This chapter covers the basic types that are used frequently in PL/SQL programs.

A scalar data type holds a single value and has no internal components. Scalar data types can be classified into four categories: number, character, date, and Boolean. Character and number data types have subtypes that associate a base type to a constraint. For example, `INTEGER` and `POSITIVE` are subtypes of the `NUMBER` base type.

For more information and the complete list of scalar data types, refer to the *PL/SQL User's Guide and Reference*.

Base Scalar Data Types

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

Base Scalar Data Types

Data Type	Description
`CHAR [(`*`maximum_length`*`)]`	Base type for fixed-length character data up to 32,767 bytes. If you do not specify a maximum length, the default length is set to 1.
`VARCHAR2` (*maximum_length*)	Base type for variable-length character data up to 32,767 bytes. There is no default size for `VARCHAR2` variables and constants.
`NUMBER` [(*precision, scale*)]	Number having precision *p* and scale *s*. The precision *p* can range from 1 to 38. The scale *s* can range from –84 to 127.
`BINARY_INTEGER`	Base type for integers between –2,147,483,647 and 2,147,483,647.

Base Scalar Data Types (continued)

Data Type	Description
PLS_INTEGER	Base type for signed integers between –2,147,483,647 and 2,147,483,647. PLS_INTEGER values require less storage and are faster than NUMBER values. In Oracle Database 10*g*, the PLS_INTEGER and BINARY_INTEGER data types are identical. The arithmetic operations on PLS_INTEGER and BINARY_INTEGER values are faster than on NUMBER values.
BOOLEAN	Base type that stores one of the three possible values used for logical calculations: TRUE, FALSE, and NULL.
BINARY_FLOAT	New data type introduced in Oracle Database 10*g*. Represents floating-point number in IEEE 754 format. Requires 5 bytes to store the value.
BINARY_DOUBLE	New data type introduced in Oracle Database 10*g*. Represents floating-point number in IEEE 754 format. Requires 9 bytes to store the value.

Base Scalar Data Types

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

ORACLE

Base Scalar Data Types (continued)

Data Type	Description
`DATE`	Base type for dates and times. `DATE` values include the time of day in seconds since midnight. The range for dates is between 4712 B.C. and 9999 A.D.
`TIMESTAMP`	The `TIMESTAMP` data type, which extends the `DATE` data type, stores the year, month, day, hour, minute, second, and fraction of second. The syntax is `TIMESTAMP[(precision)]`, where the optional parameter `precision` specifies the number of digits in the fractional part of the seconds field. You cannot use a symbolic constant or variable to specify the precision; you must use an integer literal in the range 0 ... 9. The default is 6.
`TIMESTAMP WITH TIME ZONE`	The `TIMESTAMP WITH TIME ZONE` data type, which extends the `TIMESTAMP` data type, includes a time-zone displacement. The time-zone displacement is the difference (in hours and minutes) between local time and Coordinated Universal Time (UTC), formerly known as Greenwich Mean Time. The syntax is `TIMESTAMP[(precision)] WITH TIME ZONE`, where the optional parameter `precision` specifies the number of digits in the fractional part of the seconds field. You cannot use a symbolic constant or variable to specify the precision; you must use an integer literal in the range 0 ... 9. The default is 6.

Base Scalar Data Types (continued)

Data Type	Description
TIMESTAMP WITH LOCAL TIME ZONE	The TIMESTAMP WITH LOCAL TIME ZONE data type, which extends the TIMESTAMP data type, includes a time-zone displacement. The time-zone displacement is the difference (in hours and minutes) between local time and Coordinated Universal Time (UTC), formerly known as Greenwich Mean Time. The syntax is TIMESTAMP[(precision)] WITH LOCAL TIME ZONE, where the optional parameter precision specifies the number of digits in the fractional part of the seconds field. You cannot use a symbolic constant or variable to specify the precision; you must use an integer literal in the range 0 ... 9. The default is 6. This data type differs from TIMESTAMP WITH TIME ZONE in that when you insert a value into a database column, the value is normalized to the database time zone, and the time-zone displacement is not stored in the column. When you retrieve the value, the Oracle server returns the value in your local session time zone.
INTERVAL YEAR TO MONTH	You use the INTERVAL YEAR TO MONTH data type to store and manipulate intervals of years and months. The syntax is INTERVAL YEAR[(precision)] TO MONTH, where precision specifies the number of digits in the years field. You cannot use a symbolic constant or variable to specify the precision; you must use an integer literal in the range 0 … 4. The default is 2.
INTERVAL DAY TO SECOND	You use the INTERVAL DAY TO SECOND data type to store and manipulate intervals of days, hours, minutes, and seconds. The syntax is INTERVAL DAY[(precision1)] TO SECOND[(precision2)], where precision1 and precision2 specify the number of digits in the days field and seconds field, respectively. In both cases, you cannot use a symbolic constant or variable to specify the precision; you must use an integer literal in the range 0 ... 9. The defaults are 2 and 6, respectively.

BINARY_FLOAT and BINARY_DOUBLE

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

BINARY_FLOAT and BINARY_DOUBLE

BINARY_FLOAT and BINARY_DOUBLE are new data types introduced in Oracle database 10*g*.

- **Represent floating point numbers in IEEE 754 format:** You can use these data types for scientific calculations and also for data exchange between programs that follow the IEEE (Institute of Electrical and Electronics Engineers) format.
- **Benefits:** Many computer systems support IEEE 754 floating-point operations through native processor instructions. These types are efficient for intensive computations involving floating-point data. Interaction with such programs is made easier because Oracle supports the same format to which these two data types adhere.
- **Better interoperability and operational speed:** Interoperability is mainly due to the format of these two data types. These data types improve performance in number-crunching operations such as processing scientific data.
- **Store values beyond Oracle NUMBER:** BINARY_FLOAT requires 5 bytes and BINARY_DOUBLE requires 9 bytes as opposed to Oracle NUMBER, which uses anywhere between 1 and 22 bytes. These data types meet the demand for a numeric data type that can store numeric data beyond the range of NUMBER.

BINARY_FLOAT and BINARY_DOUBLE (continued)

- **Closed arithmetic operations and transparent rounding:** All arithmetic operations with BINARY_FLOAT and BINARY_DOUBLE are closed; that is, an arithmetic operation produces a normal or special value. You need not worry about explicit conversion. For example, multiplying a BINARY_FLOAT number with another BINARY_FLOAT results in a BINARY_FLOAT number. Dividing a BINARY_FLOAT by zero is undefined and actually results in the special value Inf (Infinite). Operations on these data types are subject to rounding, which is transparent to PL/SQL users. The default mode is rounding to the nearest binary place. Most financial applications require decimal rounding behavior, whereas purely scientific applications may not.

Example

```
SET SERVEROUTPUT ON
DECLARE
  bf_var BINARY_FLOAT;
  bd_var BINARY_DOUBLE;
BEGIN
  bf_var := 270/35f;
  bd_var := 140d/0.35;
  DBMS_OUTPUT.PUT_LINE('bf: '|| bf_var);
  DBMS_OUTPUT.PUT_LINE('bd: '|| bd_var);
END;
/
```

```
bf: 7.71428585E+000
bd: 4.0E+002
PL/SQL procedure successfully completed.
```

Declaring Scalar Variables

Examples

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

ORACLE

Declaring Scalar Variables

The examples of variable declaration shown in the slide are defined as follows.

- **`emp_job`:** Variable to store an employee job title
- **`count_loop`:** Variable to count the iterations of a loop; initialized to `0`
- **`dept_total_sal`:** Variable to accumulate the total salary for a department; initialized to `0`
- **`orderdate`:** Variable to store the ship date of an order; initialized to one week from today
- **`c_tax_rate`:** Constant variable for the tax rate (which never changes throughout the PL/SQL block); set to `8.25`
- **`valid`:** Flag to indicate whether a piece of data is valid or invalid; initialized to `TRUE`

`%TYPE` Attribute

The `%TYPE` attribute

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

`%TYPE` Attribute

PL/SQL variables are usually declared to hold and manipulate data stored in a database. When you declare PL/SQL variables to hold column values, you must ensure that the variable is of the correct data type and precision. If it is not, a PL/SQL error occurs during execution. If you have to design large subprograms, this can be time consuming and error prone.

Rather than hard-coding the data type and precision of a variable, you can use the `%TYPE` attribute to declare a variable according to another previously declared variable or database column. The `%TYPE` attribute is most often used when the value stored in the variable is derived from a table in the database. When you use the `%TYPE` attribute to declare a variable, you should prefix it with the database table and column name. If you refer to a previously declared variable, prefix the variable name to the attribute.

%TYPE Attribute (continued)

Advantages of the %TYPE Attribute

- You can avoid errors caused by data type mismatch or wrong precision.
- You can avoid hard-coding the data type of a variable.
- You need not change the variable declaration if the column definition changes. If you have already declared some variables for a particular table without using the %TYPE attribute, the PL/SQL block may throw errors if the column for which the variable is declared is altered. When you use the %TYPE attribute, PL/SQL determines the data type and size of the variable when the block is compiled. This ensures that such a variable is always compatible with the column that is used to populate it.

Declaring Variables with the %TYPE Attribute

Syntax

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

Examples

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


Declaring Variables with the %TYPE Attribute

Declare variables to store the last name of an employee. The variable emp_lname is defined to be of the same data type as the last_name column in the employees table. The %TYPE attribute provides the data type of a database column.

Declare variables to store the balance of a bank account, as well as the minimum balance, which is 1,000. The variable min_balance is defined to be of the same data type as the variable balance. The %TYPE attribute provides the data type of a variable.

A NOT NULL database column constraint does not apply to variables that are declared using %TYPE. Therefore, if you declare a variable using the %TYPE attribute that uses a database column defined as NOT NULL, you can assign the NULL value to the variable.

Declaring Boolean Variables

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

Declaring Boolean Variables

With PL/SQL, you can compare variables in both SQL and procedural statements. These comparisons, called Boolean expressions, consist of simple or complex expressions separated by relational operators. In a SQL statement, you can use Boolean expressions to specify the rows in a table that are affected by the statement. In a procedural statement, Boolean expressions are the basis for conditional control. NULL stands for a missing, inapplicable, or unknown value.

Examples

```
emp_sal1 := 50000;
emp_sal2 := 60000;
```

The following expression yields TRUE:

```
emp_sal1 < emp_sal2
```

Declare and initialize a Boolean variable:

```
DECLARE
 flag BOOLEAN := FALSE;
BEGIN
 flag := TRUE;
END;
/
```

Bind Variables

Bind variables are:

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

Bind Variables

Bind variables are variables that you create in a host environment. For this reason, they are sometimes called *host variables.*

Uses of Bind Variables

Bind variables are created in the environment and not in the declarative section of a PL/SQL block. Variables declared in a PL/SQL block are available only when you execute the block. After the block is executed, the memory used by the variable is freed. However, bind variables are accessible even after the block is executed. When created, therefore, bind variables can be used and manipulated by multiple subprograms. They can be used in SQL statements and PL/SQL blocks just like any other variable. These variables can be passed as run-time values into or out of PL/SQL subprograms.

Creating Bind Variables

To create a bind variable in *i*SQL*Plus or in SQL*Plus, use the `VARIABLE` command. For example, you declare a variable of type `NUMBER` and `VARCHAR2` as follows:

```
VARIABLE return_code NUMBER
VARIABLE return_msg  VARCHAR2(30)
```

Both SQL*Plus and *i*SQL*Plus can reference the bind variable, and *i*SQL*Plus can display its value through the SQL*Plus `PRINT` command.

Bind Variables (continued)

Example

You can reference a bind variable in a PL/SQL program by preceding the variable with a colon:

```
VARIABLE result NUMBER
BEGIN
  SELECT (SALARY*12) + NVL(COMMISSION_PCT,0) INTO :result
  FROM employees WHERE employee_id = 144;
END;
/
PRINT result
```

RESULT
30000

Note: If you are creating a bind variable of type `NUMBER`, you cannot specify the precision and scale. However, you can specify the size for character strings. An Oracle `NUMBER` is stored in the same way regardless of the dimension. The Oracle server uses the same number of bytes to store 7, 70, and .0734. It is not practical to calculate the size of the Oracle number representation from the number format, so the code always allocates the bytes needed. With character strings, the size is required from the user so that the required number of bytes can be allocated.

Printing Bind Variables from the Environment

To display the current value of bind variables in the *i*SQL*Plus environment, use the `PRINT` command. However, `PRINT` cannot be used inside a PL/SQL block because it is an *i*SQL*Plus command. Note how the variable `result` is printed using the `PRINT` command in the code block shown above.

Printing Bind Variables

Example

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


Printing Bind Variables

In *i*SQL*Plus, you can display the value of a bind variable by using the `PRINT` command. When you execute the PL/SQL block shown in the slide, you see the following output when the `PRINT` command executes.

EMP_SALARY
7000

`emp_salary` is a bind variable. You can now use this variable in any SQL statement or PL/SQL program. Note the SQL statement that uses the bind variable. The output of the SQL statement is:

FIRST_NAME	LAST_NAME
Oliver	Tuvault
Sarath	Sewall
Kimberely	Grant

Note: To display all bind variables, use the `PRINT` command without a variable.

Printing Bind Variables

Example

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


Printing Bind Variables (continued)

Use the `SET AUTOPRINT ON` command to automatically display the bind variables used in a successful PL/SQL block.

Substitution Variables

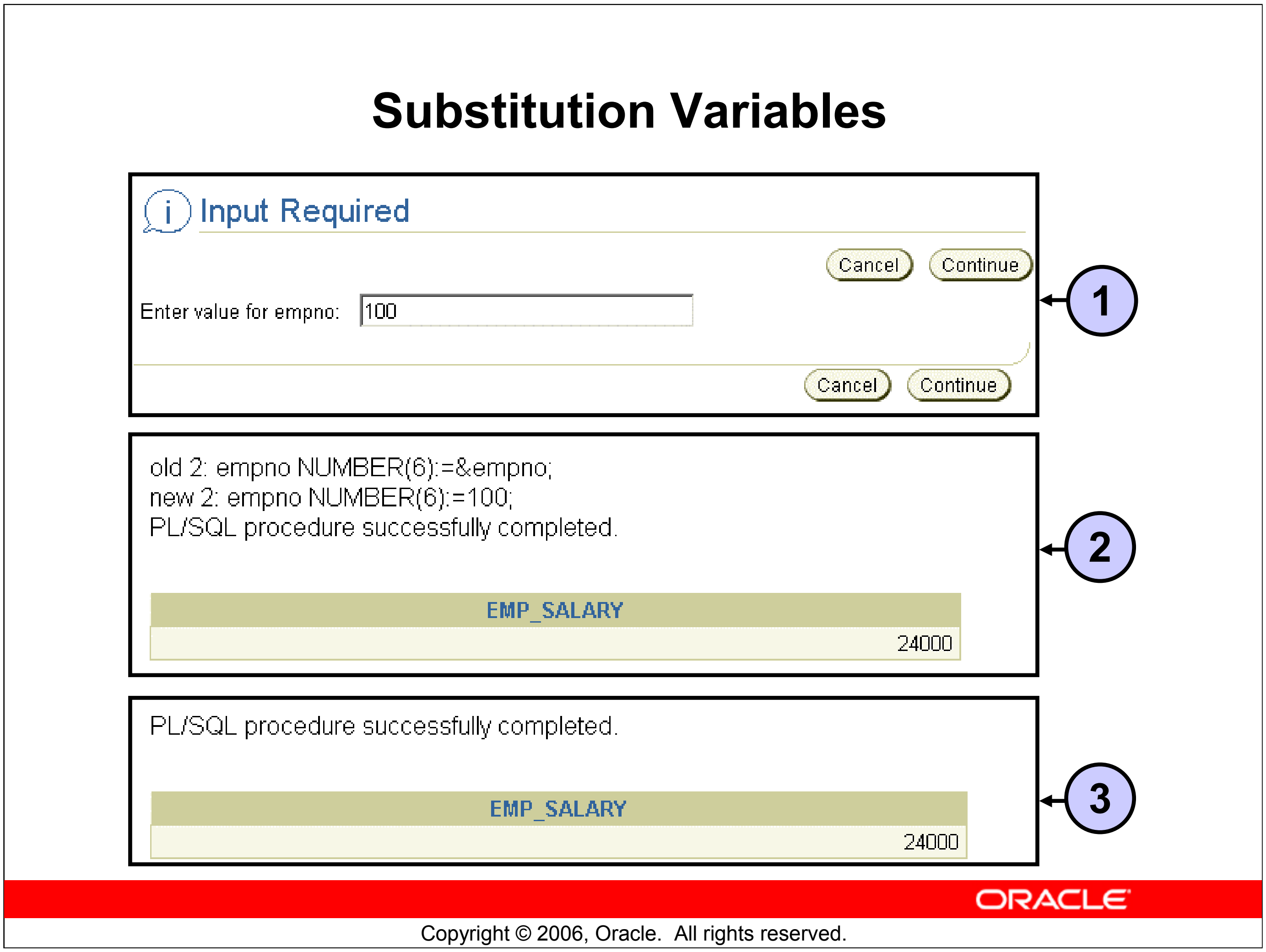
- Are used to get user input at run time
- Are referenced within a PL/SQL block with a preceding ampersand
- Are used to avoid hard-coding values that can be obtained at run time

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


Substitution Variables

In the *i*SQL*Plus environment, *i*SQL*Plus substitution variables can be used to pass run-time values into a PL/SQL block. You can reference substitution variables in SQL statements (and within a PL/SQL block) with a preceding ampersand. The text values are substituted into the PL/SQL block before the PL/SQL block is executed. Therefore, you cannot substitute different values for the substitution variables by using a loop. Even if you include the variable in a loop, you are prompted only once to enter the value. Only one value will replace the substitution variable.

When you execute the block in the slide, *i*SQL*Plus prompts you to enter a value for `empno`, which is the substitution variable.

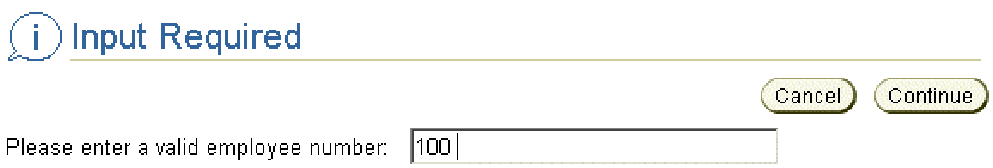


Substitution Variables (continued)

1. When you execute the block in the previous slide, *i*SQL*Plus prompts you to enter a value for `empno`, which is the substitution variable. By default, the prompt message is "Enter value for <*substitution variable*>." Enter a value as shown in the slide and click the Continue button.
2. You see the output shown in the slide. Note that *i*SQL*Plus prints both the old value and the new value for the substitution variable. You can disable this behavior by using the `SET VERIFY OFF` command.
3. This is the output after using the `SET VERIFY OFF` command.

Prompt for Substitution Variables

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



Prompt for Substitution Variables

The default prompt message in the preceding slide was “Enter value for <*substitution variable*>.”

Use the PROMPT command to change the message (as shown in this slide). This is an *i*SQL*Plus command and therefore cannot be included in the PL/SQL block.

Using DEFINE for a User Variable

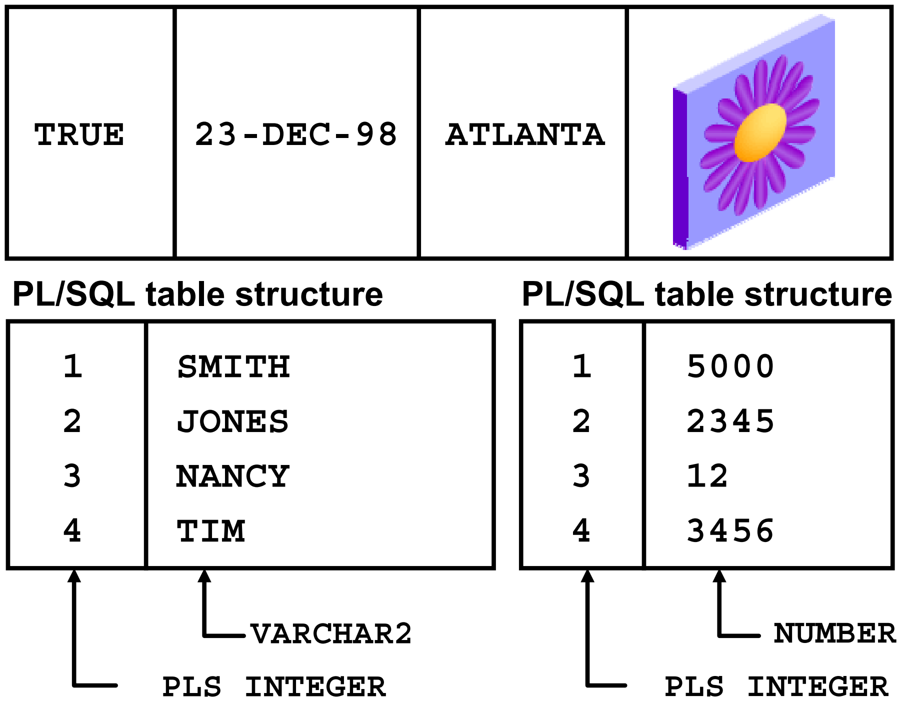
Example

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


Using `DEFINE` for a User Variable

The `DEFINE` command specifies a user variable and assigns it a `CHAR` value. You can define variables of `CHAR` data type only. Even though you enter the number `50000`, *i*SQL*Plus assigns a `CHAR` value to a variable consisting of the characters `5`,`0`,`0`,`0`, and `0`. You can reference such variables with a preceding ampersand (&), as shown in the slide.

Composite Data Types

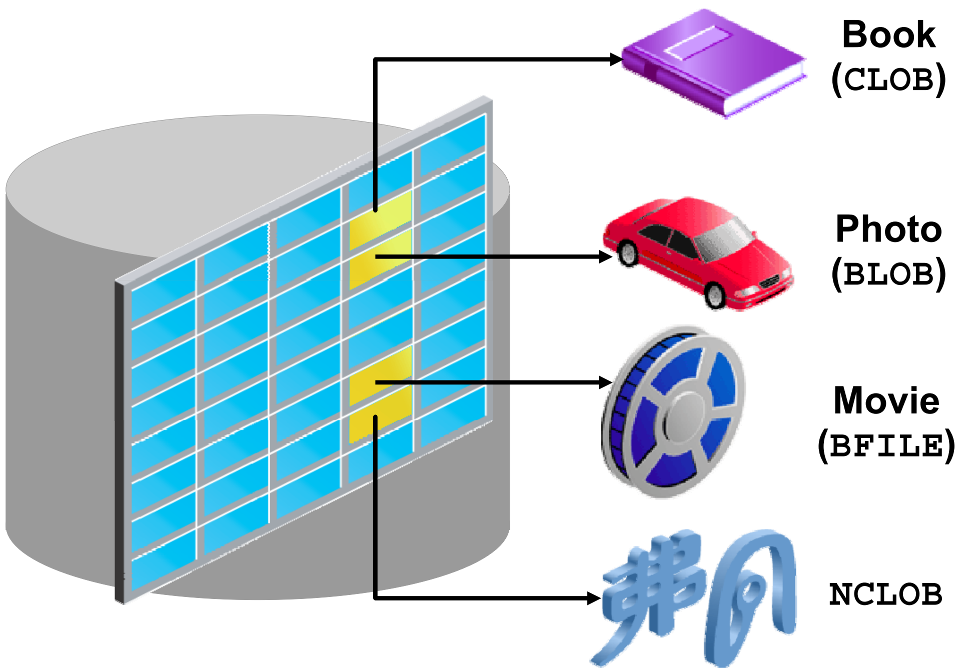


Composite Data Types

A scalar type has no internal components. A composite type has internal components that can be manipulated individually. Composite data types (also known as *collections*) are of `TABLE`, `RECORD`, `NESTED TABLE`, and `VARRAY` types.

Use the `TABLE` data type to reference and manipulate collections of data as a whole object. Use the `RECORD` data type to treat related but dissimilar data as a logical unit. `NESTED TABLE` and `VARRAY` data types are covered in the *Oracle Database 10g: Develop PL/SQL Program Units* course.

LOB Data Type Variables



LOB Data Type Variables

Large objects (LOBs) are meant to store a large amount of data. A database column can be of the LOB category. With the LOB category of data types (BLOB, CLOB, and so on), you can store blocks of unstructured data (such as text, graphic images, video clips, and sound wave forms) up to 4 GB in size. LOB data types allow efficient, random, piecewise access to the data and can be attributes of an object type.

- The character large object (CLOB) data type is used to store large blocks of character data in the database.
- The binary large object (BLOB) data type is used to store large unstructured or structured binary objects in the database. When you insert or retrieve such data to and from the database, the database does not interpret the data. External applications that use this data must interpret the data.
- The binary file (BFILE) data type is used to store large binary files. Unlike other LOBS, BFILES are not stored in the database. BFILEs are stored outside the database. They could be operating system files. Only a pointer to the BFILE is stored in the database.
- The national language character large object (NCLOB) data type is used to store large blocks of single-byte or fixed-width multibyte NCHAR unicode data in the database.

Summary

In this lesson, you should have learned how to:

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

Summary

An anonymous PL/SQL block is a basic, unnamed unit of a PL/SQL program. It consists of a set of SQL or PL/SQL statements to perform a logical function. The declarative part is the first part of a PL/SQL block and is used for declaring objects such as variables, constants, cursors, and definitions of error situations called *exceptions*.

In this lesson, you learned how to declare variables in the declarative section. You saw some of the guidelines for declaring variables. You learned how to initialize variables when you declare them.

The executable part of a PL/SQL block is the mandatory part and contains SQL and PL/SQL statements for querying and manipulating data. You learned how to initialize variables in the executable section and also how to utilize them and manipulate the values of variables.

Practice 2: Overview

This practice covers the following topics:

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

Practice 2: Overview

Exercises 1, 2, and 3 are paper based.

Practice 2

Note: It is recommended to use *i*SQL*Plus for this practice.

1. Identify valid and invalid identifier names:
   a. `today`
   b. `last_name`
   c. `today's_date`
   d. `Number_of_days_in_February_this_year`
   e. `Isleap$year`
   f. `#number`
   g. `NUMBER#`
   h. `number1to7`

2. Identify valid and invalid variable declaration and initialization:
   a. `number_of_copies PLS_INTEGER;`
   b. printer_name constant `VARCHAR2(10);`
   c. `deliver_to` `VARCHAR2(10)`:=Johnson;
   d. `by_when` `DATE:= SYSDATE+1;`

3. Examine the following anonymous block and choose the appropriate statement.

```
SET SERVEROUTPUT ON
DECLARE
  fname VARCHAR2(20);
  lname VARCHAR2(15) DEFAULT 'fernandez';
BEGIN
  DBMS_OUTPUT.PUT_LINE( FNAME ||' ' ||lname);
END;
/
```

   a. The block will execute successfully and print 'fernandez'
   b. The block will give an error because the `fname` variable is used without initializing.
   c. The block will execute successfully and print 'null fernandez'
   d. The block will give an error because you cannot use the `DEFAULT` keyword to initialize a variable of type `VARCHAR2`.
   e. The block will give an error because the variable `FNAME` is not declared.

4. Create an anonymous block. In *i*SQL*Plus, load the script `lab_01_02_soln.sql`, which you created in question 2 of practice 1.
   a. Add a declarative section to this PL/SQL block. In the declarative section, declare the following variables:
      1. Variable `today` of type `DATE`. Initialize `today` with `SYSDATE`.
      2. Variable `tomorrow` of type `today`. Use `%TYPE` attribute to declare this variable.
   b. In the executable section initialize the variable `tomorrow` with an expression, which calculates tomorrow's date (add one to the value in `today`). Print the value of `today` and `tomorrow` after printing 'Hello World'

Practice 2 (continued)

c. Execute and save this script as `lab_02_04_soln.sql`. Sample output is shown below.

```
Hello World
TODAY IS : 12-JAN-04
TOMORROW IS : 13-JAN-04
PL/SQL procedure successfully completed.
```

5. Edit the `lab_02_04_soln.sql` script.
   a. Add code to create two bind variables.
      Create bind variables `basic_percent` and `pf_percent` of type `NUMBER`.
   b. In the executable section of the PL/SQL block assign the values 45 and 12 to `basic_percent` and `pf_percent` respectively.
   c. Terminate the PL/SQL block with "/" and display the value of the bind variables by using the `PRINT` command.
   d. Execute and save your script file as `lab_02_05_soln.sql`. Sample output is shown below.

```
Hello World
TODAY IS : 12-JAN-04
TOMORROW IS : 13-JAN-04
PL/SQL procedure successfully completed.
```

BASIC_PERCENT
45

Next Page

Click the Next Page button.

PF_PERCENT
12

3

Writing Executable Statements

Objectives

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

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

Lesson Aim

You have learned how to declare variables and write executable statements in a PL/SQL block. In this lesson, you learn how lexical units make up a PL/SQL block. You learn to write nested blocks. You also learn about the scope and visibility of variables in the nested blocks and about qualifying them with labels.

Lexical Units in a PL/SQL Block

Lexical units:

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

Lexical Units in a PL/SQL Block

Lexical units include letters, numerals, special characters, tabs, spaces, returns, and symbols.

- **Identifiers:** Identifiers are the names given to PL/SQL objects. You have learned to identify valid and invalid identifiers. Recall that keywords cannot be used as identifiers.

  Quoted Identifiers:
  - Make identifiers case sensitive
  - Include characters such as spaces
  - Use reserved words

  Examples:

  ```
  "begin date" DATE;
  "end date"   DATE;
  "exception thrown" BOOLEAN DEFAULT TRUE;
  ```

  All subsequent usage of these variables should have double quotation marks.
- **Delimiters:** Delimiters are symbols that have special meaning. You have already learned that the semicolon (`;`) is used to terminate a SQL or PL/SQL statement. Therefore, `;` is the best example of a delimiter.

  For more information, please refer to the *PL/SQL User's Guide and Reference*.

Lexical Units in a PL/SQL Block (continued)

- **Delimiters (continued)**
  Delimiters are simple or compound symbols that have special meaning in PL/SQL.
  Simple Symbols

Symbol	Meaning
+	Addition operator
-	Subtraction/negation operator
*	Multiplication operator
/	Division operator
=	Equality operator
@	Remote access indicator
;	Statement terminator

Compound Symbols

Symbol	Meaning
<>	Inequality operator
!=	Inequality operator
||	Concatenation operator
--	Single-line comment indicator
/*	Beginning comment delimiter
*/	Ending comment delimiter
:=	Assignment operator

Note: This is only a subset and not a complete list of delimiters.

- **Literals:** Any value that is assigned to a variable is a literal. Any character, numeral, Boolean, or date value that is not an identifier is a literal. Literals are classified as:
  - Character literals: All string literals have the data type `CHAR` and are therefore called character literals (for example, `John`, `12C`, `1234`, and `12-JAN-1923`).
  - Numeric literals: A numeric literal represents an integer or real value (for example, `428` and `1.276`).
  - Boolean literals: Values that are assigned to Boolean variables are Boolean literals. `TRUE`, `FALSE`, and `NULL` are Boolean literals or keywords.
- **Comments:** It is good programming practice to explain what a piece of code is trying to achieve. When you include the explanation in a PL/SQL block, the compiler cannot interpret these instructions. There should be a way in which you can indicate that these instructions need not be compiled. Comments are mainly used for this purpose. Any instruction that is commented is not interpreted by the compiler.
  - Two hyphens (**`--`**) are used to comment a single line.
  - The beginning and ending comment delimiters (`/*` and `*/`) are used to comment multiple lines.

PL/SQL Block Syntax and Guidelines

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

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

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

PL/SQL Block Syntax and Guidelines

A literal is an explicit numeric, character string, date, or Boolean value that is not represented by an identifier.

- Character literals include all the printable characters in the PL/SQL character set: letters, numerals, spaces, and special symbols.
- Numeric literals can be represented either by a simple value (for example, `-32.5`) or in scientific notation (for example, `2E5` means $2 * 10^5 = 200{,}000$).

Commenting Code

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

Example

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

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


Commenting Code

You should comment code to document each phase and to assist debugging. Comment the PL/SQL code with two hyphens (`--`) if the comment is on a single line, or enclose the comment between the symbols `/*` and `*/` if the comment spans several lines.

Comments are strictly informational and do not enforce any conditions or behavior on logic or data. Well-placed comments are extremely valuable for code readability and future code maintenance. In the example in the slide, the lines enclosed within `/*` and `*/` indicate a comment that explains the following code.

SQL Functions in PL/SQL

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

SQL Functions in PL/SQL

SQL provides a number of predefined functions that can be used in SQL statements. Most of these functions are valid in PL/SQL expressions.

The following functions are not available in procedural statements:

- `DECODE`
- Group functions: `AVG`, `MIN`, `MAX`, `COUNT`, `SUM`, `STDDEV`, and `VARIANCE`
  Group functions apply to groups of rows in a table and therefore are available only in SQL statements in a PL/SQL block.

The functions mentioned here are only a subset of the complete list.

SQL Functions in PL/SQL: Examples

- **Get the length of a string:**

```
desc_size INTEGER(5);
prod_description VARCHAR2(70):='You can use this
product with your radios for higher frequency';

-- get the length of the string in prod_description
desc_size:= LENGTH(prod_description);
```

- **Convert the employee name to lowercase:**

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


SQL Functions in PL/SQL: Examples

SQL functions help you to manipulate data. They are grouped into the following categories:

- Number
- Character
- Conversion
- Date
- Miscellaneous

Data Type Conversion

- Convert data to comparable data types
- Are of two types:
  - Implicit conversions
  - Explicit conversions
- Some conversion functions:
  - `TO_CHAR`
  - `TO_DATE`
  - `TO_NUMBER`
  - `TO_TIMESTAMP`

Data Type Conversion

In any programming language, converting one data type to another is a common requirement. PL/SQL can handle such conversions with scalar data types. Data type conversions can be of two types:

Implicit conversions: PL/SQL attempts to convert data types dynamically if they are mixed in a statement. Consider the following example:

```
DECLARE
 salary NUMBER(6):=6000;
 sal_hike VARCHAR2(5):='1000';
 total_salary salary%TYPE;
BEGIN
 total_salary:=salary+sal_hike;
END;
/
```

In the example shown, the variable `sal_hike` is of type `VARCHAR2`. While calculating the total salary, PL/SQL first converts `sal_hike` to `NUMBER` and then performs the operation. The result is of the `NUMBER` type.

Implicit conversions can be between:

- Characters and numbers
- Characters and dates

Data Type Conversion (continued)

Explicit conversions: To convert values from one data type to another, use built-in functions. For example, to convert a CHAR value to a DATE or NUMBER value, use TO_DATE or TO_NUMBER, respectively.

Data Type Conversion

1
```
date_of_joining DATE:= '02-Feb-2000';
```

2
```
date_of_joining DATE:= 'February 02,2000';
```

3
```
date_of_joining DATE:= TO_DATE('February
02,2000','Month DD, YYYY');
```


Data Type Conversion (continued)

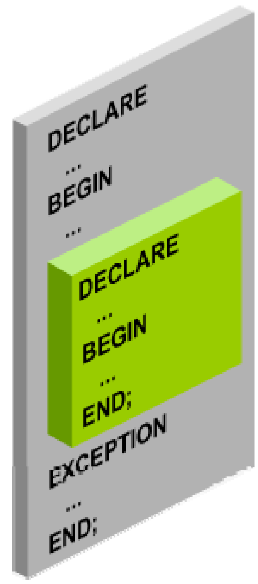
Implicit and explicit conversions of the DATE data type:

1. This example of implicit conversion assigns the date date_of_joining.
2. PL/SQL gives you an error because the date that is being assigned is not in the default format.
3. Use the TO_DATE function to explicitly convert the given date in a particular format and assign it to the DATE data type variable date_of_joining.

Nested Blocks

PL/SQL blocks can be nested.

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



Nested Blocks

One of the advantages of PL/SQL (compared to SQL) is the ability to nest statements. You can nest blocks wherever an executable statement is allowed, thus making the nested block a statement. If your executable section has code for many logically related functionalities to support multiple business requirements, you can divide the executable section into smaller blocks. The exception section can also contain nested blocks.

Nested Blocks

Example

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


Nested Blocks (continued)

The example shown in the slide has an outer (parent) block and a nested (child) block. The variable `outer_variable` is declared in the outer block and the variable `inner_variable` is declared in the inner block.

`outer_variable` is local to the outer block but global to the inner block. When you access this variable in the inner block, PL/SQL first looks for a local variable in the inner block with that name. There is no variable with the same name in the inner block, so PL/SQL looks for the variable in the outer block. Therefore, `outer_variable` is considered the global variable for all the enclosing blocks. You can access this variable in the inner block as shown in the slide. Variables declared in a PL/SQL block are considered local to that block and global to all its subblocks.

The `inner_variable` variable is local to the inner block and is not global because the inner block does not have any nested blocks. This variable can be accessed only within the inner block. If PL/SQL does not find the variable declared locally, it looks upward in the declarative section of the parent blocks. PL/SQL does not look downward in the child blocks.

Variable Scope and Visibility

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


Variable Scope and Visibility

The output of the block shown in the slide is as follows:

```
Father's Name: Patrick
Date of Birth: 12-DEC-02
Child's Name: Mike
Date of Birth: 20-APR-72
PL/SQL procedure successfully completed.
```

Examine the date of birth that is printed for father and child.

The *scope* of a variable is the portion of the program in which the variable is declared and is accessible.

The *visibility* of a variable is the portion of the program where the variable can be accessed without using a qualifier.

Scope

- The variables `father_name` and `date_of_birth` are declared in the outer block. These variables have the scope of the block in which they are declared and accessible. Therefore, the scope of these variables is limited to the outer block.

Variable Scope and Visibility (continued)

Scope (continued)

- The variables `child_name` and `date_of_birth` are declared in the inner block or the nested block. These variables are accessible only within the nested block and are not accessible in the outer block. When a variable is out of scope, PL/SQL frees the memory used to store the variable; therefore, these variables cannot be referenced.

Visibility

- The `date_of_birth` variable declared in the outer block has the scope even in the inner block. However, this variable is not visible in the inner block because the inner block has a local variable with the same name.
    1. Examine the code in the executable section of the PL/SQL block. You can print the father's name, the child's name, and the date of birth. Only the child's date of birth can be printed here because the father's date of birth is not visible.
    2. The father's date of birth is visible here and therefore can be printed.

You cannot have variables with the same name in a block. However, you can declare variables with the same name in two different blocks (nested blocks). The two items represented by the identifiers are distinct; changes in one do not affect the other.

Qualify an Identifier

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


Qualify an Identifier

A qualifier is a label given to a block. You can use a qualifier to access the variables that have scope but are not visible. Examine the code: You can now print the father's date of birth and the child's date of birth in the inner block. The outer block is labeled `outer`. You can use this label to access the `date_of_birth` variable declared in the outer block.

Because labeling is not limited to the outer block, you can label any block. The output of the code in the slide is the following:

```
Father's Name: Patrick
Date of Birth: 20-APR-72
Child's Name: Mike
Date of Birth: 12-DEC-02
PL/SQL procedure successfully completed.
```

Determining Variable Scope

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


Determining Variable Scope

Evaluate the PL/SQL block in the slide. Determine each of the following values according to the rules of scoping:

1. Value of `MESSAGE` at position 1
2. Value of `TOTAL_COMP` at position 2
3. Value of `COMM` at position 1
4. Value of `outer.COMM` at position 1
5. Value of `COMM` at position 2
6. Value of `MESSAGE` at position 2

Operators in PL/SQL

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

$\left.\right\}$ Same as in SQL

- Exponential operator (`**`)

Operators in PL/SQL

The operations in an expression are performed in a particular order depending on their precedence (priority). The following table shows the default order of operations from high priority to low priority:

Operator	Operation
`**`	Exponentiation
`+, -`	Identity, negation
`*, /`	Multiplication, division
`+, -, ||`	Addition, subtraction, concatenation
`=, <, >, <=, >=, <>, !=, ~=, ^=,` `IS NULL, LIKE, BETWEEN, IN`	Comparison
`NOT`	Logical negation
`AND`	Conjunction
`OR`	Inclusion

Operators in PL/SQL

Examples

- **Increment the counter for a loop.**

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

- **Set the value of a Boolean flag.**

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

- **Validate whether an employee number contains a value.**

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


Operators in PL/SQL (continued)

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

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

Programming Guidelines

Make code maintenance easier by:

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

Programming Guidelines

Follow programming guidelines shown in the slide to produce clear code and reduce maintenance when developing a PL/SQL block.

Code Conventions

The following table provides guidelines for writing code in uppercase or lowercase characters to help distinguish keywords from named objects.

Category	Case Convention	Examples
SQL statements	Uppercase	`SELECT`, `INSERT`
PL/SQL keywords	Uppercase	`DECLARE`, `BEGIN`, `IF`
Data types	Uppercase	`VARCHAR2`, `BOOLEAN`
Identifiers and parameters	Lowercase	`v_sal`, `emp_cursor`, `g_sal`, `p_empno`
Database tables and columns	Lowercase	`employees`, `employee_id`, `department_id`

Indenting Code

For clarity, indent each level of code.

Example:

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

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


Indenting Code

For clarity and enhanced readability, indent each level of code. To show structure, you can divide lines by using carriage returns and you can indent lines by using spaces and tabs. Compare the following IF statements for readability:

```
IF x>y THEN max:=x;ELSE max:=y;END IF;
```

```
IF x > y THEN
  max := x;
ELSE
  max := y;
END IF;
```

Summary

In this lesson, you should have learned how to:

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

Summary

Because PL/SQL is an extension of SQL, the general syntax rules that apply to SQL also apply to PL/SQL.

A block can have any number of nested blocks defined within its executable part. Blocks defined within a block are called subblocks. You can nest blocks only in the executable part of a block. Because the exception section is also in the executable section, you can have nested blocks in that section. Ensure correct scope and visibility of the variables when you have nested blocks. Avoid using the same identifiers in the parent and child blocks.

Most of the functions available in SQL are also valid in PL/SQL expressions. Conversion functions convert a value from one data type to another. Comparison operators compare one expression to another. The result is always `TRUE`, `FALSE`, or `NULL`. Typically, you use comparison operators in conditional control statements and in the `WHERE` clause of SQL data manipulation statements. The relational operators enable you to compare arbitrarily complex expressions.

Practice 3: Overview

This practice covers the following topics:

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

Practice 3: Overview

Exercises 1 and 2 are paper based.

Practice 3

Note: It is recommended to use *i*SQL*Plus for this practice.

PL/SQL Block

```
      DECLARE
       weight    NUMBER(3) := 600;
       message   VARCHAR2(255) := 'Product 10012';
      BEGIN
        DECLARE
         weight   NUMBER(3) := 1;
         message   VARCHAR2(255) := 'Product 11001';
         new_locn  VARCHAR2(50) := 'Europe';
        BEGIN
         weight := weight + 1;
1        new_locn := 'Western ' || new_locn;

        END;
       weight := weight + 1;
       message := message || ' is in stock';
2      new_locn := 'Western ' || new_locn;

      END;
      /
```

1. Evaluate the PL/SQL block given above and determine the data type and value of each of the following variables according to the rules of scoping.
   a. The value of `weight` at position 1 is:

   b. The value of `new_locn` at position 1 is:

   c. The value of `weight` at position 2 is:

   d. The value of `message` at position 2 is:

   e. The value of `new_locn` at position 2 is:

Practice 3 (continued)

Scope Example

```
DECLARE
   customer         VARCHAR2(50) := 'Womansport';
   credit_rating   VARCHAR2(50) := 'EXCELLENT';
BEGIN
    DECLARE
       customer NUMBER(7) := 201;
       name      VARCHAR2(25) := 'Unisports';
    BEGIN
       credit_rating :='GOOD';
       …
    END;
  …
END;
/
```

2. In the PL/SQL block shown above, determine the values and data types for each of the following cases.
   a. The value of `customer` in the nested block is:

   b. The value of `name` in the nested block is:

   c. The value of `credit_rating` in the nested block is:

   d. The value of `customer` in the main block is:

   e. The value of `name` in the main block is:

   f. The value of `credit_rating` in the main block is:

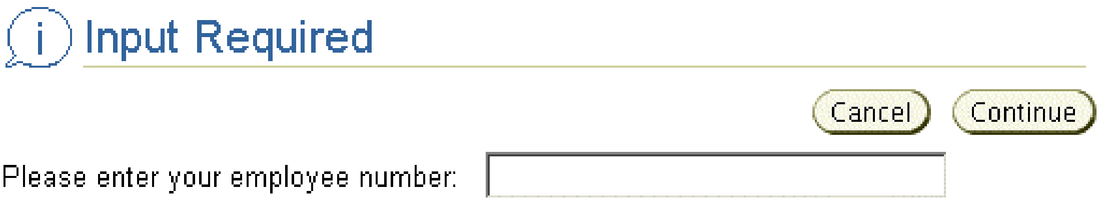
Practice 3 (continued)

3. Use the same session that you used to execute the practices in Lesson 2. If you have opened a new session, then execute `lab_02_05_soln.sql`. Edit `lab_02_05_soln.sql`.
   a. Use single line comment syntax to comment the lines that create the bind variables.
   b. Use multiple line comments in the executable section to comment the lines that assign values to the bind variables.
   c. Declare two variables: `fname` of type `VARCHAR2` and size 15, and `emp_sal` of type `NUMBER` and size 10.
   d. Include the following SQL statement in the executable section:
   ```
   SELECT first_name, salary
     INTO fname, emp_sal FROM employees
     WHERE employee_id=110;
   ```
   e. Change the line that prints 'Hello World' to print 'Hello' and the first name. You can comment the lines that display the dates and print the bind variables, if you want to.
   f. Calculate the contribution of the employee towards provident fund (PF).
   PF is 12% of the basic salary and basic salary is 45% of the salary. Use the bind variables for the calculation. Try and use only one expression to calculate the PF. Print the employee's salary and his contribution towards PF.
   g. Execute and save your script as `lab_03_03_soln.sql`. Sample output is shown below.

   ```
   Hello John
   YOUR SALARY IS : 8200
   YOUR CONTRIBUTION TOWARDS PF: 442.8
   PL/SQL procedure successfully completed.
   ```

4. Accept a value at run time using the substitution variable. In this practice, you will modify the script that you created in exercise 3 to accept user input.
   a. Load the script `lab_03_04.sql` file.
   b. Include the `PROMPT` command to prompt the user with the following message: 'Please enter your employee number.'
   c. Modify the declaration of the `empno` variable to accept the user input.
   d. Modify the select statement to include the variable `empno`.
   e. Execute and save your script as `lab_03_04_soln.sql`. Sample output is shown below.

Practice 3 (continued)



Enter 100 and click the Continue button.

```
Hello Steven
YOUR SALARY IS : 24000
YOUR CONTRIBUTION TOWARDS PF: 1296
PL/SQL procedure successfully completed.
```

5. Execute the script `lab_03_05.sql`. This script creates a table called `employee_details`.
   a. The `employee` and `employee_details` tables have the same data. You will update the data in the `employee_details` table. Do not update or change the data in the `employees` table.
   b. Open the script `lab_03_05b.sql` and observe the code in the file. Note that the code accepts the employee number and the department number from the user.
   c. You will use this as the skeleton script to develop the application, which was discussed in the lesson titled "Introduction."

4

Interacting with the Oracle Server

ORACLE

Objectives

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

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

Lesson Aim

In this lesson, you learn to embed standard SQL `SELECT`, `INSERT`, `UPDATE`, `DELETE`, and `MERGE` statements in PL/SQL blocks. You learn how to include data definition language (DDL) and transaction control statements in PL/SQL. You learn the need for cursors and differentiate between the two types of cursors. The lesson also presents the various SQL cursor attributes that can be used with implicit cursors.

SQL Statements in PL/SQL

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

SQL Statements in PL/SQL

In a PL/SQL block, you use SQL statements to retrieve and modify data from the database table. PL/SQL supports data manipulation language (DML) and transaction control commands. You can use DML commands to modify the data in a database table. However, remember the following points while using DML statements and transaction control commands in PL/SQL blocks:

- The keyword `END` signals the end of a PL/SQL block, not the end of a transaction. Just as a block can span multiple transactions, a transaction can span multiple blocks.
- PL/SQL does not directly support data definition language (DDL) statements, such as `CREATE TABLE`, `ALTER TABLE`, or `DROP TABLE`. PL/SQL supports early binding; as a result, compilation time is greater than execution time. If applications have to create database objects at run time by passing values, then early binding cannot happen in such cases. DDL statements cannot be directly executed. These statements are dynamic SQL statements. Dynamic SQL statements are built as character strings at run time and can contain placeholders for parameters. Therefore, you can use dynamic SQL to execute your DDL statements in PL/SQL. Use the `EXECUTE IMMEDIATE` statement, which takes the SQL statement as an argument to execute your DDL statement. The `EXECUTE IMMEDIATE` statement parses and executes a dynamic SQL statement.

SQL Statements in PL/SQL (continued)

Consider the following example:

```
BEGIN
CREATE TABLE My_emp_table AS SELECT * FROM employees;
END;
/
```

The example uses a DDL statement directly in the block. When you execute the block, you see the following error:

```
create table My_table as select * from table_name; * ERROR
at line 5:
ORA-06550: line 5, column 1:
PLS-00103: Encountered the symbol "CREATE" when expecting
one of the following:
…
```

Use the `EXECUTE IMMEDIATE` statement to avoid the error:

```
BEGIN
EXECUTE IMMEDIATE 'CREATE TABLE My_emp_table AS SELECT *
FROM employees';
END;
/
```

- PL/SQL does not support data control language (DCL) statements such as `GRANT` or `REVOKE`. You can use `EXECUTE IMMEDIATE` statement to execute them.
- You use transaction control statements to make the changes to the database permanent or to discard them. `COMMIT`, `ROLLBACK`, and `SAVEPOINT` are three main transactional control statements that are used. `COMMIT` is used to make the database changes permanent. `ROLLBACK` is for discarding any changes that were made to the database after the last `COMMIT`. `SAVEPOINT` is used to mark an intermediate point in transaction processing. The transaction control commands are valid in PL/SQL and therefore can be directly used in the executable section of a PL/SQL block.

SELECT Statements in PL/SQL

Retrieve data from the database with a SELECT statement.

Syntax:

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


SELECT Statements in PL/SQL

Use the SELECT statement to retrieve data from the database.

select_list	List of at least one column; can include SQL expressions, row functions, or group functions
variable_name	Scalar variable that holds the retrieved value
record_name	PL/SQL record that holds the retrieved values
table	Specifies the database table name
condition	Is composed of column names, expressions, constants, and comparison operators, including PL/SQL variables and constants

Guidelines for Retrieving Data in PL/SQL

- Terminate each SQL statement with a semicolon (;).
- Every value retrieved must be stored in a variable using the INTO clause.
- The WHERE clause is optional and can be used to specify input variables, constants, literals, and PL/SQL expressions. However, when you use the INTO clause, you should fetch only one row; using the WHERE clause is required in such cases.

`SELECT` Statements in PL/SQL (continued)

- Specify the same number of variables in the `INTO` clause as the number of database columns in the `SELECT` clause. Be sure that they correspond positionally and that their data types are compatible.
- Use group functions, such as `SUM`, in a SQL statement, because group functions apply to groups of rows in a table.

SELECT Statements in PL/SQL

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

Example

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


SELECT Statements in PL/SQL (continued)

INTO Clause

The INTO clause is mandatory and occurs between the SELECT and FROM clauses. It is used to specify the names of variables that hold the values that SQL returns from the SELECT clause. You must specify one variable for each item selected, and the order of the variables must correspond with the items selected.

Use the INTO clause to populate either PL/SQL variables or host variables.

Queries Must Return Only One Row

SELECT statements within a PL/SQL block fall into the ANSI classification of embedded SQL, for which the following rule applies: queries must return only one row. A query that returns more than one row or no row generates an error.

PL/SQL manages these errors by raising standard exceptions, which you can handle in the exception section of the block with the NO_DATA_FOUND and TOO_MANY_ROWS exceptions. Include a WHERE condition in the SQL statement so that the statement returns a single row. You learn about exception handling later in the course.

`SELECT` Statements in PL/SQL (continued)

How to Retrieve Multiple Rows from a Table and Operate on the Data

A `SELECT` statement with the `INTO` clause can retrieve only one row at a time. If your requirement is to retrieve multiple rows and operate on the data, you can make use of explicit cursors. You learn about cursors later in this lesson.

Retrieving Data in PL/SQL

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

Example

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


Retrieving Data in PL/SQL

In the example in the slide, the variables `emp_hiredate` and `emp_salary` are declared in the declarative section of the PL/SQL block. In the executable section, the values of the columns `hire_date` and `salary` for the employee with the `employee_id` 100 are retrieved from the `employees` table; they are stored in the `emp_hiredate` and `emp_salary` variables, respectively. Observe how the `INTO` clause, along with the `SELECT` statement, retrieves the database column values into the PL/SQL variables.

Note: The `SELECT` statement retrieves `hire_date` and then `salary`. The variables in the `INTO` clause must thus be in the same order. For example, if you exchange `emp_hiredate` and `emp_salary` in the statement in the slide, the statement results in an error.

Retrieving Data in PL/SQL

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

Example

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


Retrieving Data in PL/SQL (continued)

In the example in the slide, the `sum_sal` and `deptno` variables are declared in the declarative section of the PL/SQL block. In the executable section, the total salary for the employees in the department with the `department_id` 60 is computed using the SQL aggregate function `SUM`. The calculated total salary is assigned to the `sum_sal` variable.

Note: Group functions cannot be used in PL/SQL syntax. They are used in SQL statements within a PL/SQL block as shown in the example. You cannot use them as follows:

```
sum_sal := SUM(employees.salary);
```

The output of the PL/SQL block in the slide is the following:

```
The sum of salary is 28800
PL/SQL procedure successfully completed.
```

Naming Conventions

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

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


Naming Conventions

In potentially ambiguous SQL statements, the names of database columns take precedence over the names of local variables.

The example shown in the slide is defined as follows: Retrieve the hire date and today's date from the `employees` table for `employee_id` 176. This example raises an unhandled run-time exception because in the `WHERE` clause, the PL/SQL variable names are the same as the database column names in the `employees` table.

The following `DELETE` statement removes all employees from the `employees` table where the last name is not null (not just "King") because the Oracle server assumes that both occurrences of `last_name` in the `WHERE` clause refer to the database column:

```
DECLARE
   last_name VARCHAR2(25) := 'King';
BEGIN
   DELETE FROM employees WHERE last_name = last_name;
. . .
```

Naming Conventions

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

Naming Conventions (continued)

Avoid ambiguity in the WHERE clause by adhering to a naming convention that distinguishes database column names from PL/SQL variable names.

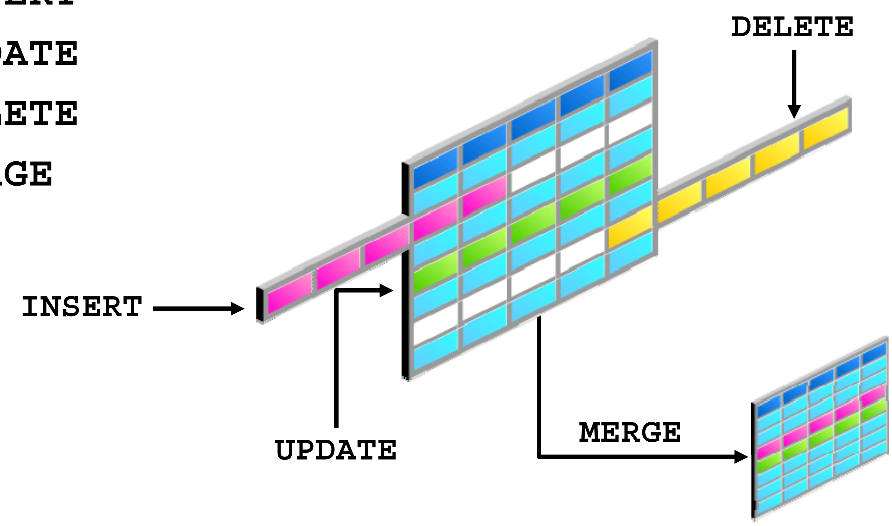
- Database columns and identifiers should have distinct names.
- Syntax errors can arise because PL/SQL checks the database first for a column in the table.

Note: There is no possibility for ambiguity in the SELECT clause because any identifier in the SELECT clause must be a database column name. There is no possibility for ambiguity in the INTO clause because identifiers in the INTO clause must be PL/SQL variables. There is the possibility of confusion only in the WHERE clause.

Manipulating Data Using PL/SQL

Make changes to database tables by using DML commands:

- `INSERT`
- `UPDATE`
- `DELETE`
- `MERGE`



Manipulating Data Using PL/SQL

You manipulate data in the database by using the DML commands. You can issue the DML commands `INSERT`, `UPDATE`, `DELETE` and `MERGE` without restriction in PL/SQL. Row locks (and table locks) are released by including `COMMIT` or `ROLLBACK` statements in the PL/SQL code.

- The `INSERT` statement adds new rows to the table.
- The `UPDATE` statement modifies existing rows in the table.
- The `DELETE` statement removes rows from the table.
- The `MERGE` statement selects rows from one table to update or insert into another table. The decision whether to update or insert into the target table is based on a condition in the `ON` clause.

Note: `MERGE` is a deterministic statement. That is, you cannot update the same row of the target table multiple times in the same `MERGE` statement. You must have `INSERT` and `UPDATE` object privileges in the target table and the `SELECT` privilege on the source table.

Inserting Data

Add new employee information to the `EMPLOYEES` table.

Example

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


Inserting Data

In the example in the slide, an `INSERT` statement is used within a PL/SQL block to insert a record into the `employees` table. While using the `INSERT` command in a PL/SQL block, you can:

- Use SQL functions, such as `USER` and `SYSDATE`
- Generate primary key values by using existing database sequences
- Derive values in the PL/SQL block

Note: The data in the `employees` table needs to remain unchanged. Inserting, updating, and deleting are thus not allowed on this table.

Updating Data

Increase the salary of all employees who are stock clerks.

Example

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


Updating Data

There may be ambiguity in the `SET` clause of the `UPDATE` statement because, although the identifier on the left of the assignment operator is always a database column, the identifier on the right can be either a database column or a PL/SQL variable. Recall that if column names and identifier names are identical in the `WHERE` clause, the Oracle server looks to the database first for the name.

Remember that the `WHERE` clause is used to determine which rows are affected. If no rows are modified, no error occurs (unlike the `SELECT` statement in PL/SQL).

Note: PL/SQL variable assignments always use `:=`, and SQL column assignments always use `=`.

Deleting Data

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

Example

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


Deleting Data

The DELETE statement removes unwanted rows from a table. If the WHERE clause is not used, all the rows in a table can be removed if there are no integrity constraints.

Merging Rows

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

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


Merging Rows

The `MERGE` statement inserts or updates rows in one table by using data from another table. Each row is inserted or updated in the target table depending on an equijoin condition.

The example shown matches the `employee_id` in the `COPY_EMP` table to the `employee_id` in the `employees` table. If a match is found, the row is updated to match the row in the `employees` table. If the row is not found, it is inserted into the `copy_emp` table.

The complete example for using `MERGE` in a PL/SQL block is shown on the next notes page.

Merging Rows (continued)

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

SQL Cursor

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

SQL Cursor

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

Where Does Oracle Process SQL Statements?

The Oracle server allocates a private memory area called the *context area* for processing SQL statements. The SQL statement is parsed and processed in this area. Information required for processing and information retrieved after processing is all stored in this area. You have no control over this area because it is internally managed by the Oracle server.

A cursor is a pointer to the context area. However, this cursor is an implicit cursor and is automatically managed by the Oracle server. When the executable block issues a SQL statement, PL/SQL creates an implicit cursor.

There are two types of cursors:

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

SQL Cursor (continued)

- **Explicit:** 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. 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.

SQL Cursor Attributes for Implicit Cursors

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

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

SQL Cursor Attributes for Implicit Cursors

SQL cursor attributes enable you to evaluate what happened when an implicit cursor was last used. Use these attributes in PL/SQL statements but not in SQL statements.

You can test the attributes `SQL%ROWCOUNT`, `SQL%FOUND`, and `SQL%NOTFOUND` in the executable section of a block to gather information after the appropriate DML command. PL/SQL does not return an error if a DML statement does not affect rows in the underlying table. However, if a `SELECT` statement does not retrieve any rows, PL/SQL returns an exception.

Observe that the attributes are prefixed with SQL. These cursor attributes are used with implicit cursors that are automatically created by PL/SQL and for which you do not know the names. Therefore, you use SQL instead of the cursor name.

The `SQL%NOTFOUND` attribute is opposite to `SQL%FOUND`. This attribute may be used as the exit condition in a loop. It is useful in `UPDATE` and `DELETE` statements when no rows are changed because exceptions are not returned in these cases.

You learn about explicit cursor attributes later in the course.

SQL Cursor Attributes for Implicit Cursors

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

Example

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


SQL Cursor Attributes for Implicit Cursors (continued)

The example in the slide deletes a row with `employee_id` 176 from the `employees` table. Using the `SQL%ROWCOUNT` attribute, you can print the number of rows deleted.

Summary

In this lesson, you should have learned how to:

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

Summary

The DML commands and transaction control statements can be used in PL/SQL programs without restriction. However, the DDL commands cannot be used directly.

A `SELECT` statement in PL/SQL block can return only one row. It is mandatory to use the `INTO` clause to hold the values retrieved by the `SELECT` statement.

A cursor is a pointer to the memory area. There are two types of cursors. Implicit cursors are created and managed internally by the Oracle server to execute SQL statements. You can use SQL cursor attributes with these cursors to determine the outcome of the SQL statement. Explicit cursors are declared by programmers.

Practice 4: Overview

This practice covers the following topics:

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

Practice 4

Note: It is recommended to use *i*SQL*Plus for this practice.

1. Create a PL/SQL block that selects the maximum department ID in the `departments` table and stores it in the `max_deptno` variable. Display the maximum department ID.
    a. Declare a variable `max_deptno` of type `NUMBER` in the declarative section.
    b. Start the executable section with the keyword `BEGIN` and include a `SELECT` statement to retrieve the maximum `department_id` from the `departments` table.
    c. Display `max_deptno` and end the executable block.
    d. Execute and save your script as `lab_04_01_soln.sql`. Sample output is

    ```
    The maximum department_id is : 270
    PL/SQL procedure successfully completed.
    ```

2. Modify the PL/SQL block you created in exercise 1 to insert a new department into the `departments` table.
    a. Load the script `lab_04_01_soln.sql`. Declare two variables:
    `dept_name` of type `departments.department_name`.
    Bind variable `dept_id` of type `NUMBER`.
    Assign 'Education' to `dept_name` in the declarative section.
    b. You have already retrieved the current maximum department number from the `departments` table. Add 10 to it and assign the result to `dept_id`.
    c. Include an `INSERT` statement to insert data into the `department_name`, `department_id`, and `location_id` columns of the `departments` table. Use values in `dept_name`, `dept_id` for `department_name`, `department_id` and use NULL for `location_id`.
    d. Use the SQL attribute `SQL%ROWCOUNT` to display the number of rows that are affected.
    e. Execute a select statement to check if the new department is inserted. You can terminate the PL/SQL block with "/" and include the `SELECT` statement in your script.
    f. Execute and save your script as `lab_04_02_soln.sql`. Sample output is

    ```
    The maximum department_id is : 270
    SQL%ROWCOUNT gives 1
    PL/SQL procedure successfully completed.
    ```

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
280	Education		

Practice 4 (continued)

3. In exercise 2, you have set `location_id` to null. Create a PL/SQL block that updates the `location_id` to 3000 for the new department. Use the bind variable `dept_id` to update the row.
   Note: Skip step a if you have not started a new *i*SQL*Plus session for this practice.
   a. If you have started a new *i*SQL*Plus session, delete the department that you have added to the `departments` table and execute the script `lab_04_02_soln.sql`.
   b. Start the executable block with the keyword `BEGIN`. Include the `UPDATE` statement to set the `location_id` to 3000 for the new department. Use the bind variable `dept_id` in your `UPDATE` statement.
   c. End the executable block with the keyword `END`. Terminate the PL/SQL block with "/" and include a `SELECT` statement to display the department that you updated.
   d. Finally, include a `DELETE` statement to delete the department that you added.
   e. Execute and save your script as `lab_04_03_soln.sql`. Sample output is shown below.

PL/SQL procedure successfully completed.

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
280	Education		3000

1 row deleted.

4. Load the script `lab_03_05b.sql` to the *i*SQL*Plus workspace.
   a. Observe that the code has nested blocks. You will see the declarative section of the outer block. a. Look for the comment "INCLUDE EXECUTABLE SECTION OF OUTER BLOCK HERE" and start an executable section
   b. Include a single `SELECT` statement, which retrieves the `employee_id` of the employee working in the 'Human Resources' department. Use the `INTO` clause to store the retrieved value in the variable `emp_authorization`.
   c. Save your script as `lab_04_04_soln.sql`.

5

Writing Control Structures

Objectives

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

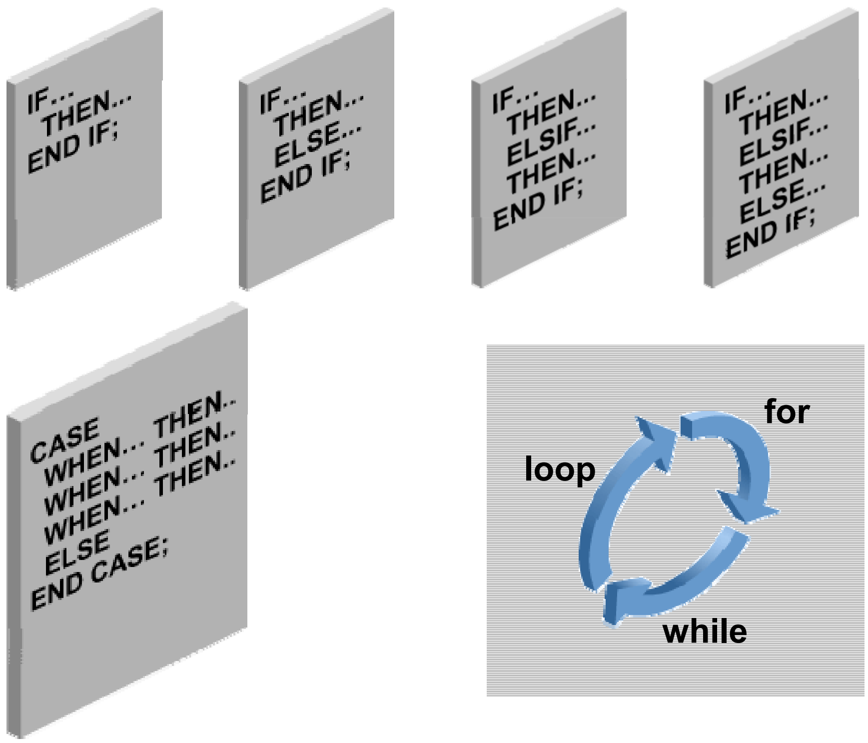
- **Identify the uses and types of control structures**
- **Construct an `IF` statement**
- **Use `CASE` statements and `CASE` expressions**
- **Construct and identify different loop statements**
- **Use guidelines when using conditional control structures**

Lesson Aim

You have learned to write PL/SQL blocks containing declarative and executable sections. You have also learned to include expressions and SQL statements in the executable block.

In this lesson, you learn how to use control structures such as `IF` statements, `CASE` expressions, and `LOOP` structures in a PL/SQL block.

Controlling Flow of Execution



Controlling Flow of Execution

You can change the logical flow of statements within the PL/SQL block with a number of control structures. This lesson addresses three types of PL/SQL control structures: conditional constructs with the `IF` statement, `CASE` expressions, and `LOOP` control structures.

IF Statements

Syntax:

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


IF Statements

The structure of the PL/SQL `IF` statement is similar to the structure of `IF` statements in other procedural languages. It allows PL/SQL to perform actions selectively based on conditions.

In the syntax:

condition	Is a Boolean variable or expression that returns `TRUE`, `FALSE`, or `NULL`
`THEN`	Introduces a clause that associates the Boolean expression with the sequence of statements that follows it
statements	Can be one or more PL/SQL or SQL statements. (They may include further `IF` statements containing several nested `IF`, `ELSE`, and `ELSIF` statements.) The statements in the `THEN` clause are executed only if the condition in the associated `IF` clause evaluates to `TRUE`.

IF Statements (continued)

In the syntax:

ELSIF	Is a keyword that introduces a Boolean expression (If the first condition yields FALSE or NULL, the ELSIF keyword introduces additional conditions.)
ELSE	Introduces the default clause that is executed if and only if none of the earlier predicates (introduced by IF and ELSIF) are TRUE. The tests are executed in sequence so that a later predicate that might be true is preempted by an earlier predicate that is true.
END IF	END IF marks the end of an IF statement

Note: ELSIF and ELSE are optional in an IF statement. You can have any number of ELSIF keywords but only one ELSE keyword in your IF statement. END IF marks the end of an IF statement and must be terminated by a semicolon.

Simple IF Statement

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

PL/SQL procedure successfully completed.

Simple IF Statement

The slide shows an example of a simple IF statement with the THEN clause. The variable myage is initialized to 31. The condition for the IF statement returns FALSE because myage is not less than 11. Therefore, the control never reaches the THEN clause. We add code to this example to see the usage of ELSE and ELSIF.

An IF statement can have multiple conditional expressions related with logical operators such as AND, OR, and NOT. Here is an example:

```
IF (myfirstname='Christopher' AND myage <11)
…
```

The condition uses the AND operator and therefore evaluates to TRUE only if both conditions are evaluated as TRUE. There is no limitation on the number of conditional expressions. However, these statements must be related with appropriate logical operators.

IF THEN ELSE Statement

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

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

ORACLE

IF THEN ELSE Statement

An ELSE clause is added to the code in the previous slide. The condition has not changed and therefore still evaluates to FALSE. Recall that the statements in the THEN clause are executed only if the condition returns TRUE. In this case, the condition returns FALSE and the control moves to the ELSE statement. The output of the block is shown in the slide.

IF ELSIF ELSE Clause

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

```
I am in my thirties
PL/SQL procedure successfully completed.
```


IF ELSIF ELSE Clause

The `IF` clause now contains multiple `ELSIF` clauses and an `ELSE`. Notice that the `ELSIF` clauses can have conditions, unlike the `ELSE` clause. The condition for `ELSIF` should be followed by the `THEN` clause, which is executed if the condition of the `ELSIF` returns `TRUE`.

When you have multiple `ELSIF` clauses, if the first condition is `FALSE` or `NULL`, the control shifts to the next `ELSIF` clause. Conditions are evaluated one by one from the top. If all conditions are `FALSE` or `NULL`, the statements in the `ELSE` clause are executed. The final `ELSE` clause is optional.

NULL Values in IF Statements

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

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


NULL Values in IF Statements

In the example shown in the slide, the variable `myage` is declared but not initialized. The condition in the `IF` statement returns `NULL` rather than `TRUE` or `FALSE`. In such a case, the control goes to the `ELSE` statement.

Guidelines:

- You can perform actions selectively based on conditions that are being met.
- When writing code, remember the spelling of the keywords:
  - `ELSIF` is one word
  - `END IF` is two words
- If the controlling Boolean condition is `TRUE`, the associated sequence of statements is executed; if the controlling Boolean condition is `FALSE` or `NULL`, the associated sequence of statements is passed over. Any number of `ELSIF` clauses are permitted.
- Indent the conditionally executed statements for clarity.

CASE Expressions

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

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


CASE Expressions

A CASE expression returns a result based on one or more alternatives. To return the result, the CASE expression uses a *selector*, which is an expression whose value is used to return one of several alternatives. The selector is followed by one or more WHEN clauses that are checked sequentially. The value of the selector determines which result is returned. If the value of the selector equals the value of a WHEN clause expression, that WHEN clause is executed and that result is returned.

PL/SQL also provides a searched CASE expression, which has the form:

```
CASE
   WHEN search_condition1 THEN result1
   WHEN search_condition2 THEN result2
   ...
   WHEN search_conditionN THEN resultN
  [ELSE resultN+1]
END;
```

A searched CASE expression has no selector. Furthermore, its WHEN clauses contain search conditions that yield a Boolean value rather than expressions that can yield a value of any type.

CASE Expressions: Example

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


CASE Expressions: Example

In the example in the slide, the `CASE` expression uses the value in the `grade` variable as the expression. This value is accepted from the user by using a substitution variable. Based on the value entered by the user, the `CASE` expression returns the value of the `appraisal` variable based on the value of the `grade` value. The output of the example is as follows when you enter `a` or `A` for the grade:

```
Grade: A Appraisal Excellent
PL/SQL procedure successfully completed.
```

Searched CASE Expressions

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


Searched CASE Expressions

In the previous example, you saw a single test expression that was the `grade` variable. The `WHEN` clause compared a value against this test expression.

In searched `CASE` statements, you do not have a test expression. Instead, the `WHEN` clause contains an expression that results in a Boolean value. The same example is rewritten in this slide to show searched `CASE` statements.

CASE Statement

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


CASE Statement

Recall the use of the IF statement. You may include n number of PL/SQL statements in the THEN clause and also in the ELSE clause. Similarly, you can include statements in the CASE statement. The CASE statement is more readable compared to multiple IF and ELSIF statements.

How Is a CASE Expression Different from a CASE Statement?

A CASE expression evaluates the condition and returns a value. On the other hand, a CASE statement evaluates the condition and performs an action. A CASE statement can be a complete PL/SQL block. CASE statements end with END CASE; but CASE expressions end with END;.

Handling Nulls

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

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

Handling Nulls

Consider the following example:

```
x := 5;
y := NULL;
...
IF x != y THEN  -- yields NULL, not TRUE
  --  sequence_of_statements that are not executed
END IF;
```

You may expect the sequence of statements to execute because `x` and `y` seem unequal. But nulls are indeterminate. Whether or not `x` is equal to `y` is unknown. Therefore, the `IF` condition yields `NULL` and the sequence of statements is bypassed.

```
a := NULL;
b := NULL;
...
IF a = b THEN  -- yields NULL, not TRUE
  --  sequence_of_statements that are not executed
END IF;
```

In the second example, you may expect the sequence of statements to execute because `a` and `b` seem equal. But, again, equality is unknown, so the `IF` condition yields `NULL` and the sequence of statements is bypassed.

Logic Tables

Build a simple Boolean condition with a comparison operator.

AND	*TRUE*	*FALSE*	*NULL*
TRUE	TRUE	FALSE	NULL
FALSE	FALSE	FALSE	FALSE
NULL	NULL	FALSE	NULL

OR	*TRUE*	*FALSE*	*NULL*
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	NULL
NULL	TRUE	NULL	NULL

NOT	
TRUE	FALSE
FALSE	TRUE
NULL	NULL

Logic Tables

You can build a simple Boolean condition by combining number, character, and date expressions with comparison operators.

You can build a complex Boolean condition by combining simple Boolean conditions with the logical operators AND, OR, and NOT. The logical operators are used to check the Boolean variable values and return TRUE, FALSE, or NULL. In the logic tables shown in the slide:

- FALSE takes precedence in an AND condition, and TRUE takes precedence in an OR condition
- AND returns TRUE only if both of its operands are TRUE
- OR returns FALSE only if both of its operands are FALSE
- NULL AND TRUE always evaluates to NULL because it is not known whether the second operand evaluates to TRUE or not.

Note: The negation of NULL (NOT NULL) results in a null value because null values are indeterminate.

Boolean Conditions

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

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

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

Boolean Conditions

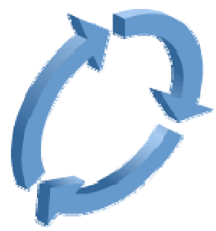
The AND logic table can help you evaluate the possibilities for the Boolean condition in the slide.

Answers

1. TRUE
2. FALSE
3. NULL
4. FALSE

Iterative Control: LOOP Statements

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



Iterative Control: LOOP Statements

PL/SQL provides a number of facilities to structure loops to repeat a statement or sequence of statements multiple times. Loops are mainly used to execute statements repeatedly until an exit condition is reached. It is mandatory to have an exit condition in a loop; otherwise, the loop is infinite.

Looping constructs are the second type of control structure. PL/SQL provides the following types of loops:

- Basic loop that performs repetitive actions without overall conditions
- `FOR` loops that perform iterative actions based on a count
- `WHILE` loops that perform iterative actions based on a condition

Note: An `EXIT` statement can be used to terminate loops. A basic loop must have an `EXIT`. The cursor `FOR LOOP` (which is another type of `FOR LOOP`) is discussed in the lesson titled “Using Explicit Cursors.”

Basic Loops

Syntax:

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


Basic Loops

The simplest form of a LOOP statement is the basic (or infinite) loop, which encloses a sequence of statements between the keywords LOOP and END LOOP. Each time the flow of execution reaches the END LOOP statement, control is returned to the corresponding LOOP statement above it. A basic loop allows execution of its statements at least once, even if the EXIT condition is already met upon entering the loop. Without the EXIT statement, the loop would be infinite.

EXIT Statement

You can use the EXIT statement to terminate a loop. Control passes to the next statement after the END LOOP statement. You can issue EXIT either as an action within an IF statement or as a stand-alone statement within the loop. The EXIT statement must be placed inside a loop. In the latter case, you can attach a WHEN clause to enable conditional termination of the loop. When the EXIT statement is encountered, the condition in the WHEN clause is evaluated. If the condition yields TRUE, the loop ends and control passes to the next statement after the loop. A basic loop can contain multiple EXIT statements, but it is recommended that you have only one EXIT point.

Basic Loops

Example

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


Basic Loops (continued)

The basic loop example shown in the slide is defined as follows: Insert three new location IDs for the CA country code and the city of Montreal.

Note: A basic loop allows execution of its statements at least once, even if the condition has been met upon entering the loop. This happens only if the condition is placed in the loop so that it is not checked until after these statements. However, if the exit condition is placed at the top of the loop (before any of the other executable statements) and if that condition is true, the loop exits and the statements never execute.

`WHILE` Loops

Syntax:

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

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

`WHILE` Loops

You can use the `WHILE` loop to repeat a sequence of statements until the controlling condition is no longer `TRUE`. The condition is evaluated at the start of each iteration. The loop terminates when the condition is `FALSE` or `NULL`. If the condition is `FALSE` or `NULL` at the start of the loop, no further iterations are performed.

In the syntax:

condition	Is a Boolean variable or expression (`TRUE`, `FALSE`, or `NULL`)
statement	Can be one or more PL/SQL or SQL statements

If the variables involved in the conditions do not change during the body of the loop, the condition remains `TRUE` and the loop does not terminate.

Note: If the condition yields `NULL`, the loop is bypassed and control passes to the next statement.

WHILE Loops

Example

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

ORACLE

WHILE Loops (continued)

In the example in the slide, three new locations IDs for the CA country code and the city of Montreal are added.

With each iteration through the WHILE loop, a counter (counter) is incremented. If the number of iterations is less than or equal to the number 3, then the code within the loop is executed and a row is inserted into the locations table. After the counter exceeds the number of new locations for this city and country, the condition that controls the loop evaluates to FALSE and the loop terminates.

FOR Loops

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

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


FOR Loops

FOR loops have the same general structure as the basic loop. In addition, they have a control statement before the LOOP keyword to set the number of iterations that PL/SQL performs. In the syntax:

counter	Is an implicitly declared integer whose value automatically increases or decreases (decreases if the REVERSE keyword is used) by 1 on each iteration of the loop until the upper or lower bound is reached
REVERSE	Causes the counter to decrement with each iteration from the upper bound to the lower bound **Note:** The lower bound is still referenced first.
lower_bound	Specifies the lower bound for the range of counter values
upper_bound	Specifies the upper bound for the range of counter values

Do not declare the counter. It is declared implicitly as an integer.

FOR Loops (continued)

Note: The sequence of statements is executed each time the counter is incremented, as determined by the two bounds. The lower bound and upper bound of the loop range can be literals, variables, or expressions, but they must evaluate to integers. The bounds are rounded to integers; that is, 11/3 and 8/5 are valid upper or lower bounds. The lower bound and upper bound are inclusive in the loop range. If the lower bound of the loop range evaluates to a larger integer than the upper bound, the sequence of statements is not executed.

For example, the following statement is executed only once:

```
FOR i IN 3..3
LOOP
 statement1;
END LOOP;
```

FOR Loops

Example

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


FOR Loops (continued)

You have already learned how to insert three new locations for the CA country code and the city Montreal by using the basic loop and the WHILE loop. This slide shows you how to achieve the same by using the FOR loop.

FOR Loops

Guidelines

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

FOR Loops (continued)

The slide lists the guidelines to follow when writing a FOR loop.

Note: The lower and upper bounds of a LOOP statement do not need to be numeric literals. They can be expressions that convert to numeric values.

Example:

```
DECLARE
  lower  NUMBER := 1;
  upper  NUMBER := 100;
BEGIN
  FOR i IN lower..upper LOOP
  ...
  END LOOP;
END;
/
```

Guidelines for Loops

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

Guidelines for Loops

A basic loop allows execution of its statement at least once, even if the condition is already met upon entering the loop. Without the `EXIT` statement, the loop would be infinite.

You can use the `WHILE` loop to repeat a sequence of statements until the controlling condition is no longer `TRUE`. The condition is evaluated at the start of each iteration. The loop terminates when the condition is `FALSE`. If the condition is `FALSE` at the start of the loop, no further iterations are performed.

`FOR` loops have a control statement before the `LOOP` keyword to determine the number of iterations that PL/SQL performs. Use a `FOR` loop if the number of iterations is predetermined.

Nested Loops and Labels

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

Nested Loops and Labels

You can nest `FOR`, `WHILE`, and basic loops within one another. The termination of a nested loop does not terminate the enclosing loop unless an exception was raised. However, you can label loops and exit the outer loop with the `EXIT` statement.

Label names follow the same rules as other identifiers. A label is placed before a statement, either on the same line or on a separate line. White space is insignificant in all PL/SQL parsing except inside literals. Label basic loops by placing the label before the word `LOOP` within label delimiters (<<*label*>>). In `FOR` and `WHILE` loops, place the label before `FOR` or `WHILE`.

If the loop is labeled, the label name can optionally be included after the `END LOOP` statement for clarity.

Nested Loops and Labels

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


Nested Loops and Labels (continued)

In the example in the slide, there are two loops. The outer loop is identified by the label `<<Outer_Loop>>` and the inner loop is identified by the label `<<Inner_Loop>>`. The identifiers are placed before the word `LOOP` within label delimiters (<<*label*>>). The inner loop is nested within the outer loop. The label names are included after the `END LOOP` statements for clarity.

Summary

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

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

Summary

A language can be called a programming language only if it provides control structures for the implementation of the business logic. These control structures are also used to control the flow of the program. PL/SQL is a programming language that integrates programming constructs with SQL.

A conditional control construct checks for the validity of a condition and performs an action accordingly. You use the `IF` construct to perform a conditional execution of statements.

An iterative control construct executes a sequence of statements repeatedly, as long as a specified condition holds `TRUE`. You use the various loop constructs to perform iterative operations.

Practice 5: Overview

This practice covers the following topics:

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

Practice 5: Overview

In this practice, you create PL/SQL blocks that incorporate loops and conditional control structures. The exercises test your understanding of writing various `IF` statements and `LOOP` constructs.

Practice 5

1. Execute the command in the file `lab_05_01.sql` to create the `messages` table. Write a PL/SQL block to insert numbers into the `messages` table.
   a. Insert the numbers 1 to 10, excluding 6 and 8.
   b. Commit before the end of the block.
   c. Execute a `SELECT` statement to verify that your PL/SQL block worked. You should see the following output.

RESULTS
1
2
3
4
5
7
9
10

8 rows selected.

2. Execute the script `lab_05_02.sql`. This script creates an `emp` table that is a replica of the `employees` table. It alters the `emp` table to add a new column, `stars`, of `VARCHAR2` data type and size 50. Create a PL/SQL block that inserts an asterisk in the `stars` column for every $1000 of the employee's salary. Save your script as `lab_05_02_soln.sql`.
   a. Use the `DEFINE` command to define a variable called `empno` and initialize it to 176.
   b. Start the declarative section of the block and pass the value of `empno` to the PL/SQL block through an *i*SQL*Plus substitution variable. Declare a variable `asterisk` of type `emp.stars` and initialize it to `NULL`. Create a variable `sal` of type `emp.salary`.
   c. In the executable section, write logic to append an asterisk (*) to the string for every $1000 of the salary amount. For example, if the employee earns $8000, the string of asterisks should contain eight asterisks. If the employee earns $12500, the string of asterisks should contain 13 asterisks.
   d. Update the `stars` column for the employee with the string of asterisks. Commit before the end of the block.

e. Display the row from the emp table to verify whether your PL/SQL block has executed successfully.

f. Execute and save your script as `lab_05_02_soln.sql`. The output is shown below.

EMPLOYEE_ID	SALARY	STARS
176	8600	*********

3. Load the script `lab_04_04_soln.sql`, which you created in question 4 of Practice 4.

   a. Look for the comment "INCLUDE SIMPLE IF STATEMENT HERE" and include a simple `IF` statement to check if the values of `emp_id` and `emp_authorization` are the same.

   b. Save your script as `lab_05_03_soln.sql`.

Working with Composite Data Types

6

Objectives

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

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

Lesson Aim

You have already been introduced to composite data types. In this lesson, you learn more about composite data types and their uses.

Composite Data Types

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

Composite Data Types

You have learned that variables of scalar data type can hold only one value, whereas a variable of composite data type can hold multiple values of scalar data type or composite data type. There are two types of composite data types:

- **PL/SQL records:** Records are used to treat related but dissimilar data as a logical unit. A PL/SQL record can have variables of different types. For example, you can define a record to hold employee details. This involves storing an employee number as `NUMBER`, a first name and last name as `VARCHAR2`, and so on. By creating a record to store employee details, you create a logical collective unit. This makes data access and manipulation easier.
- **PL/SQL collections:** Collections are used to treat data as a single unit. Collections are of three types:
  - `INDEX BY` tables or associative arrays
  - Nested table
  - `VARRAY`

Why Use Composite Data Types?

You have all the related data as a single unit. You can easily access and modify the data. Data is easier to manage, relate, and transport if it is composite. An analogy is having a single bag for all your laptop components rather than a separate bag for each component.

Composite Data Types

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

Composite Data Types (continued)

If both PL/SQL records and PL/SQL collections are composite types, how do you choose which one to use?

Use PL/SQL records when you want to store values of different data types that are logically related. If you create a record to hold employee details, indicate that all the values stored are related because they provide information about a particular employee.

Use PL/SQL collections when you want to store values of the same data type. Note that this data type can also be of the composite type (such as records). You can define a collection to hold the first names of all employees. You may have stored *n* names in the collection; however, name 1 is not related to name 2. The relation between these names is only that they are employee names. These collections are similar to arrays in programming languages such as C, C++, and Java.

PL/SQL Records

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

PL/SQL Records

A record is a group of related data items stored in fields, each with its own name and data type.

- Each record defined can have as many fields as necessary.
- Records can be assigned initial values and can be defined as `NOT NULL`.
- Fields without initial values are initialized to `NULL`.
- The `DEFAULT` keyword can also be used when defining fields.
- You can define `RECORD` types and declare user-defined records in the declarative part of any block, subprogram, or package.
- You can declare and reference nested records. One record can be the component of another record.

Creating a PL/SQL Record

Syntax:

1
```
TYPE type_name IS RECORD
     (field_declaration[, field_declaration]…);
```

2
```
identifier    type_name;
```

***field_declaration*:**

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

ORACLE

Creating a PL/SQL Record

PL/SQL records are user-defined composite types. To use them:

1. Define the record in the declarative section of a PL/SQL block. The syntax for defining the record is shown in the slide.
2. Declare (and optionally initialize) the internal components of this record type.

In the syntax:

type_name	Is the name of the RECORD type (This identifier is used to declare records.)
field_name	Is the name of a field within the record
field_type	Is the data type of the field (It represents any PL/SQL data type except REF CURSOR. You can use the %TYPE and %ROWTYPE attributes.)
expr	Is the *field_type* or an initial value

The NOT NULL constraint prevents assigning nulls to those fields. Be sure to initialize the NOT NULL fields.

REF CURSOR is covered in appendix C ("REF Cursors").

Creating a PL/SQL Record

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

Example

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


Creating a PL/SQL Record (continued)

Field declarations used in defining a record are like variable declarations. Each field has a unique name and a specific data type. There are no predefined data types for PL/SQL records, as there are for scalar variables. Therefore, you must create the record type first and then declare an identifier using that type.

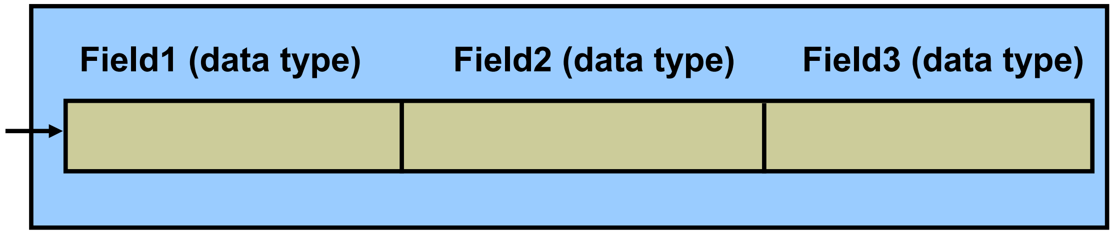
In the example in the slide, a record type (`emp_record_type`) is defined to hold the values for `last_name`, `job_id`, and `salary`. In the next step, a record (`emp_record`) of the type `emp_record_type` is declared.

The following example shows that you can use the `%TYPE` attribute to specify a field data type:

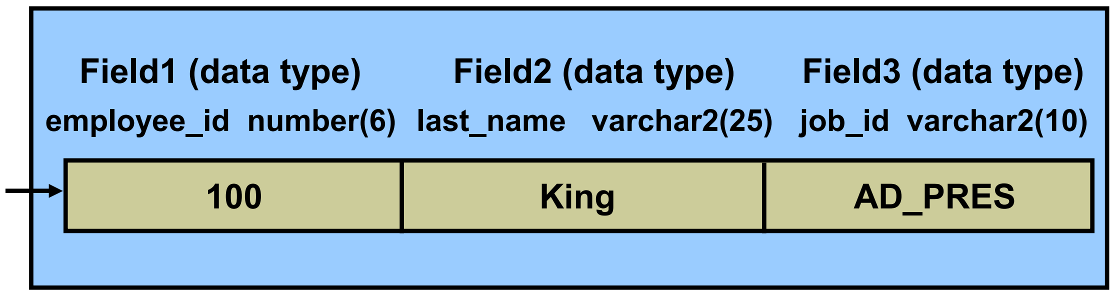
```
DECLARE
 TYPE emp_record_type IS RECORD
 (employee_id    NUMBER(6) NOT NULL := 100,
  last_name      employees.last_name%TYPE,
  job_id         employees.job_id%TYPE);
  emp_record     emp_record_type;
  ...
```

Note: You can add the `NOT NULL` constraint to any field declaration to prevent assigning nulls to that field. Remember that the fields declared as `NOT NULL` must be initialized.

PL/SQL Record Structure



Example



PL/SQL Record Structure

Fields in a record are accessed with the name of the record. To reference or initialize an individual field, use the dot notation:

```
record_name.field_name
```

For example, you reference the `job_id` field in the `emp_record` record as follows:

```
emp_record.job_id
```

You can then assign a value to the record field:

```
emp_record.job_id := 'ST_CLERK';
```

In a block or subprogram, user-defined records are instantiated when you enter the block or subprogram. They cease to exist when you exit the block or subprogram.

%ROWTYPE Attribute

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

Syntax:

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

ORACLE

%ROWTYPE Attribute

You have learned that %TYPE is used to declare a variable of a column type. The variable has the same data type and size as the table column. The benefit of %TYPE is that you do not have to change the variable if the column is altered. Also, if the variable is used in any calculations, you need not worry about its precision.

The %ROWTYPE attribute is used to declare a record that can hold an entire row of a table or view. The fields in the record take their names and data types from the columns of the table or view. The record can also store an entire row of data fetched from a cursor or cursor variable.

The slide shows the syntax for declaring a record. In the syntax:

identifier	Is the name chosen for the record as a whole
reference	Is the name of the table, view, cursor, or cursor variable on which the record is to be based (The table or view must exist for this reference to be valid.)

In the following example, a record is declared using %ROWTYPE as a data type specifier:

```
DECLARE
  emp_record   employees%ROWTYPE;
  ...
```

%ROWTYPE Attribute (continued)

The emp_record record has a structure consisting of the following fields, each representing a column in the employees table.

Note: This is not code but simply the structure of the composite variable.

```
(employee_id       NUMBER(6),
 first_name        VARCHAR2(20),
 last_name         VARCHAR2(20),
 email             VARCHAR2(20),
 phone_number      VARCHAR2(20),
 hire_date         DATE,
 salary            NUMBER(8,2),
 commission_pct    NUMBER(2,2),
 manager_id        NUMBER(6),
 department_id     NUMBER(4))
```

To reference an individual field, use dot notation:

```
record_name.field_name
```

For example, you reference the commission_pct field in the emp_record record as follows:

```
emp_record.commission_pct
```

You can then assign a value to the record field:

```
emp_record.commission_pct:= .35;
```

Assigning Values to Records

You can assign a list of common values to a record by using the SELECT or FETCH statement. Make sure that the column names appear in the same order as the fields in your record. You can also assign one record to another if both have the same corresponding data types. A user-defined record and a %ROWTYPE record never have the same data type.

Advantages of Using %ROWTYPE

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

Advantages of Using %ROWTYPE

The advantages of using the `%ROWTYPE` attribute are listed in the slide. Use the `%ROWTYPE` attribute when you are not sure about the structure of the underlying database table.

The main advantage of using `%ROWTYPE` is that it simplifies maintenance. Using `%ROWTYPE` ensures that the data types of the variables declared with this attribute change dynamically when the underlying table is altered. If a DDL statement changes the columns in a table, then the PL/SQL program unit is invalidated. When the program is recompiled, it will automatically reflect the new table format.

The `%ROWTYPE` attribute is particularly useful when you want to retrieve an entire row from a table. In the absence of this attribute, you would be forced to declare a variable for each of the columns retrieved by the select statement.

%ROWTYPE Attribute

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


%ROWTYPE Attribute

An example of the %ROWTYPE attribute is shown in the slide. If an employee is retiring, information about that employee is added to a table that holds information about retired employees. The user supplies the employee number. The record of the employee specified by the user is retrieved from the employees table and stored in the emp_rec variable, which is declared using the %ROWTYPE attribute.

The record that is inserted into the retired_emps table is shown below:

EMPNO	ENAME	JOB	MGR	HIREDATE	LEAVEDATE	SAL	COMM	DEPTNO
124	Mourgos	ST_MAN	100	16-NOV-99	26-JAN-04	5800		50

Inserting a Record by Using %ROWTYPE

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

ORACLE

Inserting a Record by Using %ROWTYPE

Compare the insert statement in the previous slide with the insert statement in this slide. The `emp_rec` record is of type `retired_emps`. The number of fields in the record must be equal to the number of field names in the `INTO` clause. You can use this record to insert values into a table. This makes the code more readable.

The create statement that creates `retired_emps` is:

```
CREATE TABLE retired_emps
   (EMPNO      NUMBER(4), ENAME      VARCHAR2(10),
    JOB        VARCHAR2(9),MGR        NUMBER(4),
    HIREDATE   DATE, LEAVEDATE  DATE,
    SAL        NUMBER(7,2), COMM       NUMBER(7,2),
    DEPTNO     NUMBER(2))
```

Examine the select statement in the slide. We select `hire_date` twice and insert the `hire_date` value in the `leavedate` field of `retired_emps`. No employee retires on the hire date. The record that is inserted is shown below. (You will see how to update this in the next slide.)

EMPNO	ENAME	JOB	MGR	HIREDATE	LEAVEDATE	SAL	COMM	DEPTNO
124	Mourgos	ST_MAN	100	16-NOV-99	16-NOV-99	5800		50

Updating a Row in a Table by Using a Record

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


Updating a Row in a Table by Using a Record

You have learned to insert a row by using a record. This slide shows you how to update a row by using a record. The keyword `ROW` is used to represent the entire row. The code shown in the slide updates the `leavedate` of the employee. The record is updated.

EMPNO	ENAME	JOB	MGR	HIREDATE	LEAVEDATE	SAL	COMM	DEPTNO
124	Mourgos	ST_MAN	100	16-NOV-99	27-JAN-04	5800		50

`INDEX BY` Tables or Associative Arrays

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

`INDEX BY` Tables or Associative Arrays

`INDEX BY` tables are composite types (collections) and are user defined. `INDEX BY` tables can store data using a primary key value as the index, where the key values are not sequential. `INDEX BY` tables are sets of key-value pairs. (You can imagine data stored in two columns, although the key and value pairs are not exactly stored in columns.)

`INDEX BY` tables have only two columns:

- A column of integer or string type that acts as the primary key. The key can be numeric, either `BINARY_INTEGER` or `PLS_INTEGER`. The `BINARY_INTEGER` and `PLS_INTEGER` keys require less storage than `NUMBER`. They are used to represent mathematical integers compactly and to implement arithmetic operations by using machine arithmetic. Arithmetic operations on these data types are faster than `NUMBER` arithmetic. The key can also be of type `VARCHAR2` or one of its subtypes. The examples in this course use the `PLS_INTEGER` data type for the key column.
- A column of scalar or record data type to hold values. If the column is of scalar type, it can hold only one value. If the column is of record type, it can hold multiple values.

The `INDEX BY` tables are unconstrained in size. However, the key in the `PLS_INTEGER` column is restricted to the maximum value that a `PLS_INTEGER` can hold. Note that the keys can be both positive and negative. The keys in `INDEX BY` tables are not in sequence.

Creating an INDEX BY Table

Syntax:

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

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

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


Creating an INDEX BY Table

There are two steps involved in creating an INDEX BY table.

1. Declare a TABLE data type.
2. Declare a variable of that data type.

In the syntax:

type_name	Is the name of the TABLE type (It is a type specifier used in subsequent declarations of PL/SQL table identifiers.)
column_type	Is any scalar or composite data type such as VARCHAR2, DATE, NUMBER, or %TYPE (You can use the %TYPE attribute to provide the column data type.)
identifier	Is the name of the identifier that represents an entire PL/SQL table

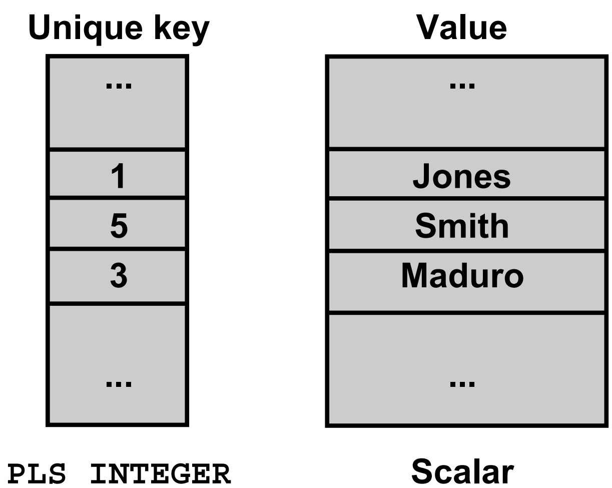
Creating an `INDEX BY` Table (continued)

The `NOT NULL` constraint prevents nulls from being assigned to the PL/SQL table of that type. Do not initialize the `INDEX BY` table.

`INDEX BY` tables can have the following element types: `BINARY_INTEGER`, `BOOLEAN`, `LONG`, `LONG RAW`, `NATURAL`, `NATURALN`, `PLS_INTEGER`, `POSITIVE`, `POSITIVEN`, `SIGNTYPE`, and `STRING`.

`INDEX BY` tables are not automatically populated when you create them. You must programmatically populate the `INDEX BY` tables in your PL/SQL programs and then use them.

INDEX BY Table Structure



INDEX BY Table Structure

Like the size of a database table, the size of an INDEX BY table is unconstrained. That is, the number of rows in an INDEX BY table can increase dynamically so that your INDEX BY table grows as new rows are added.

INDEX BY tables can have one column and a unique identifier to that column, neither of which can be named. The column can belong to any scalar or record data type, but the primary key must belong to the types PLS_INTEGER or BINARY_INTEGER. You cannot initialize an INDEX BY table in its declaration. An INDEX BY table is not populated at the time of declaration. It contains no keys or values. An explicit executable statement is required to populate the INDEX BY table.

Creating an INDEX BY Table

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


Creating an INDEX BY Table

The example in the slide creates two INDEX BY tables.

Use the key of the INDEX BY table to access an element in the table.

Syntax:

```
INDEX_BY_table_name(index)
```

Here, index belongs to type PLS_INTEGER.

The following example shows how to reference the third row in an INDEX BY table called ename_table:

```
ename_table(3)
```

The magnitude range of a PLS_INTEGER is –2147483647 to 2147483647, so the primary key value can be negative. Indexing does not need to start with 1.

Note: The exists(i) method returns TRUE if a row with index *i* is returned. Use the exists method to prevent an error that is raised in reference to a nonexistent table element.

Using `INDEX BY` Table Methods

The following methods make `INDEX BY` tables easier to use:

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

Using `INDEX BY` Table Methods

An `INDEX BY` table method is a built-in procedure or function that operates on a PL/SQL table and is called by using dot notation.

Syntax: `table_name.method_name[ (parameters) ]`

Method	Description
`EXISTS(n)`	Returns `TRUE` if the *n*th element in a PL/SQL table exists
`COUNT`	Returns the number of elements that a PL/SQL table currently contains
`FIRST` `LAST`	• Returns the first and last (smallest and largest) index numbers in a PL/SQL table • Returns `NULL` if the PL/SQL table is empty
`PRIOR(n)`	Returns the index number that precedes index *n* in a PL/SQL table
`NEXT(n)`	Returns the index number that succeeds index *n* in a PL/SQL table
`DELETE`	• `DELETE` removes all elements from a PL/SQL table. • `DELETE(n)` removes the *n*th element from a PL/SQL table. • `DELETE(m, n)` removes all elements in the range *m* ... *n* from a PL/SQL table.

`INDEX BY` Table of Records

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

Example

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


`INDEX BY` Table of Records

At any particular time, an `INDEX BY` table declared as a table of scalar data type can store the details of only one column in a database table. There is often a need to store all the columns retrieved by a query. The `INDEX BY` table of records offers a solution to this. Because only one table definition is needed to hold information about all the fields of a database table, the table of records greatly increases the functionality of `INDEX BY` tables.

Referencing a Table of Records

In the example in the slide, you can refer to fields in the `dept_table` record because each element of the table is a record.

Syntax:

```
table(index).field
```

Example:

```
dept_table(15).location_id := 1700;
```

`location_id` represents a field in `dept_table`.

Referencing a Table of Records (continued)

You can use the %ROWTYPE attribute to declare a record that represents a row in a database table. The differences between the %ROWTYPE attribute and the composite data type PL/SQL record include the following:

- PL/SQL record types can be user defined, whereas %ROWTYPE implicitly defines the record.
- PL/SQL records enable you to specify the fields and their data types while declaring them. When you use %ROWTYPE, you cannot specify the fields. The %ROWTYPE attribute represents a table row with all the fields based on the definition of that table.
- User-defined records are static. %ROWTYPE records are dynamic because the table structures are altered in the database.

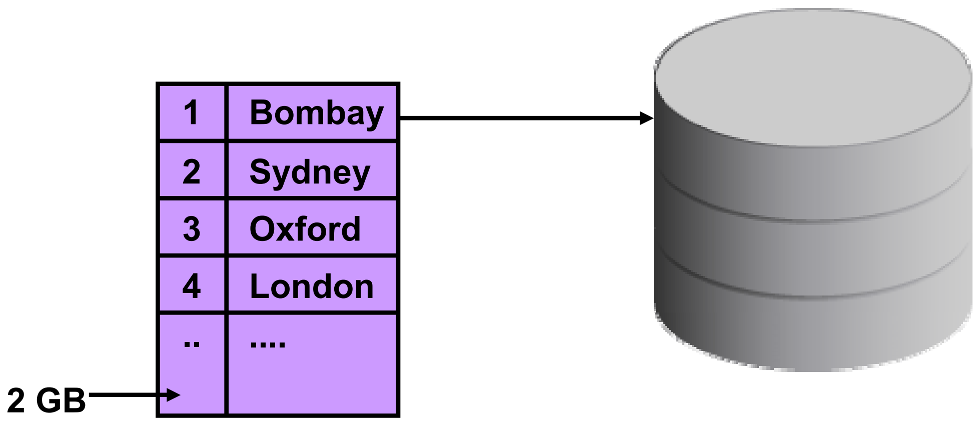
INDEX BY Table of Records: Example

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


INDEX BY Table of Records: Example

The example in the slide declares an INDEX BY table of records emp_table_type to temporarily store the details of employees whose employee IDs are between 100 and 104. Using a loop, the information of the employees from the EMPLOYEES table is retrieved and stored in the INDEX BY table. Another loop is used to print the last names from the INDEX BY table. Note the use of the first and last methods in the example.

Nested Tables



Nested Tables

The functionality of nested tables is similar to that of `INDEX BY` tables; however, there are differences in the nested table implementation. The nested table is a valid data type in a schema-level table, but an `INDEX BY` table is not. The key type for nested tables is not `PLS_INTEGER`. The key cannot be a negative value (unlike in the `INDEX BY` table). Though we are referring to the first column as key, there is no key in a nested table. There is a column with numbers in sequence that is considered as the key column. Elements can be deleted from anywhere in a nested table, leaving a sparse table with nonsequential keys. The rows of a nested table are not in any particular order. When you retrieve values from a nested table, the rows are given consecutive subscripts starting from 1. Nested tables can be stored in the database (unlike `INDEX BY` tables).

Syntax:

```
TYPE type_name IS TABLE OF
 {column_type | variable%TYPE
 | table.column%TYPE} [NOT NULL]
 | table.%ROWTYPE
```

In Oracle Database 10*g*, nested tables can be compared for equality. You can check if an element exists in a nested table and also if a nested table is a subset of another.

Nested Tables (continued)

Example:

```
TYPE location_type IS TABLE OF locations.city%TYPE;
offices location_type;
```

If you do not initialize an `INDEX BY` table, it is empty. If you do not initialize a nested table, it is automatically initialized to `NULL`. You can initialize the `offices` nested table by using a constructor:

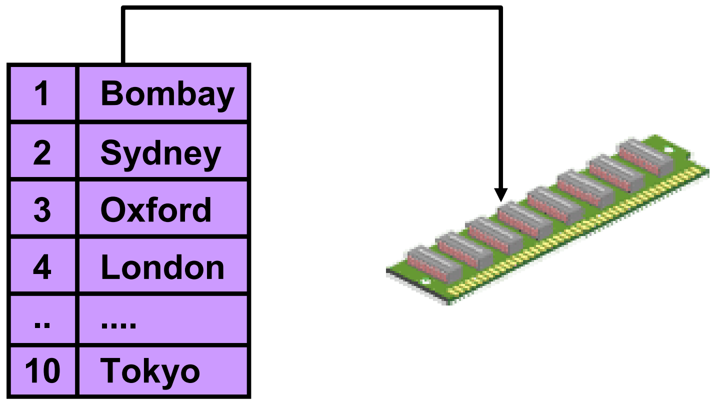
```
offices := location_type('Bombay', 'Tokyo','Singapore',
'Oxford');
```

Complete example:

```
SET SERVEROUTPUT ON
DECLARE
   TYPE location_type IS TABLE OF locations.city%TYPE;
    offices location_type;
    table_count NUMBER;
BEGIN
   offices := location_type('Bombay', 'Tokyo','Singapore',
    'Oxford');
   table_count := offices.count();
   FOR i in 1..table_count LOOP
     DBMS_OUTPUT.PUT_LINE(offices(i));
   END LOOP;
END;
/
```

```
Bombay
Tokyo
Singapore
Oxford
PL/SQL procedure successfully completed.
```

VARRAY



VARRAY

A variable-size array (VARRAY) is similar to a PL/SQL table, except that a VARRAY is constrained in size. VARRAY is valid in a schema-level table. Items of VARRAY type are called VARRAYs. VARRAYs have a fixed upper bound. You have to specify the upper bound when you declare them. This is similar to arrays in the C language. The maximum size of a VARRAY is 2 GB, as in nested tables. The distinction between a nested table and a VARRAY is the physical storage mode. The elements of a VARRAY are stored contiguously in memory and not in the database. You can create a VARRAY type in the database by using SQL.

Example:

```
TYPE location_type IS VARRAY(3) OF locations.city%TYPE;
offices location_type;
```

The size of this VARRAY is restricted to 3. You can initialize a VARRAY by using constructors. If you try to initialize the VARRAY with more than three elements, a "Subscript outside of limit" error message is displayed.

Summary

In this lesson, you should have learned how to:

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

Summary

A PL/SQL record is a collection of individual fields that represent a row in the table. By using records, you can group the data into one structure and then manipulate this structure as one entity or logical unit. This helps reduce coding and keeps the code easier to maintain and understand.

Like PL/SQL records, the table is another composite data type. `INDEX BY` tables are objects of `TABLE` type and look similar to database tables but with a slight difference. `INDEX BY` tables use a primary key to give you array-like access to rows. The size of an `INDEX BY` table is unconstrained. `INDEX BY` tables store a key and a value pair. The key column must be of the `PLS_INTEGER` or `BINARY_INTEGER` type; the column that holds the value can be of any data type.

The key type for nested tables is not `PLS_INTEGER`. The key cannot have a negative value, unlike the case with `INDEX BY` tables. The key must also be in a sequence.

Variable-size arrays (`VARRAYs`) are similar to PL/SQL tables, except that a `VARRAY` is constrained in size.

Practice 6: Overview

This practice covers the following topics:

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

Practice 6: Overview

In this practice, you define, create, and use `INDEX BY` tables and a PL/SQL record.

Practice 6

1. Write a PL/SQL block to print information about a given country.
   a. Declare a PL/SQL record based on the structure of the `countries` table.
   b. Use the `DEFINE` command to define a variable `countryid`. Assign CA to `countryid`. Pass the value to the PL/SQL block through an *i*SQL*Plus substitution variable.
   c. In the declarative section, use the `%ROWTYPE` attribute and declare the variable `country_record` of type `countries`.
   d. In the executable section, get all the information from the `countries` table by using `countryid`. Display selected information about the country. A sample output is shown below.

   ```
   Country Id: CA Country Name: Canada Region: 2
   PL/SQL procedure successfully completed.
   ```

   e. You may want to execute and test the PL/SQL block for the countries with the IDs DE, UK, US.
2. Create a PL/SQL block to retrieve the name of some departments from the `departments` table and print each department name on the screen, incorporating an `INDEX BY` table. Save the script as `lab_06_02_soln.sql`.
   a. Declare an `INDEX BY` table `dept_table_type` of type `departments.department_name`. Declare a variable `my_dept_table` of type `dept_table_type` to temporarily store the name of the departments.
   b. Declare two variables: `loop_count` and `deptno` of type `NUMBER`. Assign 10 to `loop_count` and 0 to `deptno`.
   c. Using a loop, retrieve the name of 10 departments and store the names in the `INDEX BY` table. Start with `department_id` 10. Increase `deptno` by 10 for every iteration of the loop. The following table shows the `department_id` for which you should retrieve the `department_name` and store in the `INDEX BY` table.

DEPARTMENT_ID	DEPARTMENT_NAME
10	Administration
20	Marketing
30	Purchasing
40	Human Resources
50	Shipping
60	IT
70	Public Relations
80	Sales
90	Executive
100	Finance

Practice 6 (continued)

d. Using another loop, retrieve the department names from the `INDEX BY` table and display them.

e. Execute and save your script as `lab_06_02_soln`.sql. The output is shown below.

```
Administration
Marketing
Purchasing
Human Resources
Shipping
IT
Public Relations
Sales
Executive
Finance
PL/SQL procedure successfully completed.
```

Practice 6 (continued)

3. Modify the block that you created in question 2 to retrieve all information about each department from the `departments` table and display the information. Use an `INDEX BY` table of records.
   a. Load the script `lab_06_02_soln.sql`.
   b. You have declared the `INDEX BY` table to be of type `departments.department_name`. Modify the declaration of the `INDEX BY` table, to temporarily store the number, name, and location of the departments. Use the `%ROWTYPE` attribute.
   c. Modify the select statement to retrieve all department information currently in the `departments` table and store it in the `INDEX BY` table.
   d. Using another loop, retrieve the department information from the `INDEX BY` table and display the information. A sample output is shown below.

```
Department Number: 10 Department Name: Administration Manager
Id: 200 Location Id: 1700
Department Number: 20 Department Name: Marketing Manager Id:
201 Location Id: 1800
Department Number: 30 Department Name: Purchasing Manager Id:
114 Location Id: 1700
Department Number: 40 Department Name: Human Resources
Manager Id: 203 Location Id: 2400
Department Number: 50 Department Name: Shipping Manager Id:
121 Location Id: 1500
Department Number: 60 Department Name: IT Manager Id: 103
Location Id: 1400
Department Number: 70 Department Name: Public Relations
Manager Id: 204 Location Id: 2700
Department Number: 80 Department Name: Sales Manager Id: 145
Location Id: 2500
Department Number: 90 Department Name: Executive Manager Id:
100 Location Id: 1700
Department Number: 100 Department Name: Finance Manager Id:
108 Location Id: 1700
PL/SQL procedure successfully completed.
```

4. Load the script `lab_05_03_soln.sql`.
   a. Look for the comment "DECLARE AN INDEX BY TABLE OF TYPE VARCHAR2(50). CALL IT ename_table_type" and include the declaration.
   b. Look for the comment "DECLARE A VARIABLE ename_table OF TYPE ename_table_type" and include the declaration.
   c. Save your script as `lab_06_04_soln.sql`.

Using Explicit Cursors

Objectives

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

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

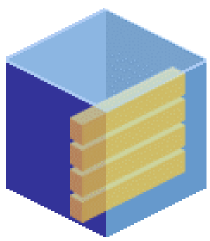
Lesson Aim

You have learned about implicit cursors that are automatically created by PL/SQL when you execute a SQL `SELECT` or DML statement. In this lesson, you learn about explicit cursors. You learn to differentiate between implicit and explicit cursors. You also learn to declare and control simple cursors as well as cursors with parameters.

Cursors

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

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



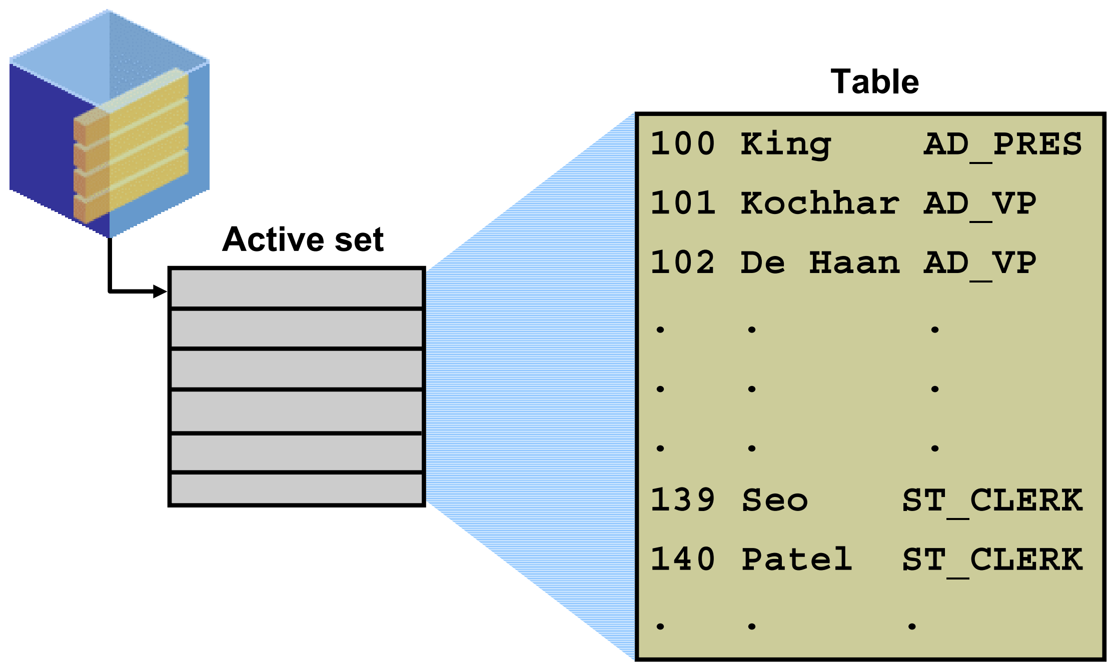
Cursors

The Oracle server uses work areas (called *private SQL areas*) to execute SQL statements and to store processing information. You can use explicit cursors to name a private SQL area and to access its stored information.

Cursor Type	Description
Implicit	Implicit cursors are declared by PL/SQL implicitly for all DML and PL/SQL `SELECT` statements.
Explicit	For queries that return more than one row, explicit cursors are declared and managed by the programmer and manipulated through specific statements in the block's executable actions.

The Oracle server implicitly opens a cursor to process each SQL statement that is not associated with an explicitly declared cursor. Using PL/SQL, you can refer to the most recent implicit cursor as the SQL cursor.

Explicit Cursor Operations



Explicit Cursor Operations

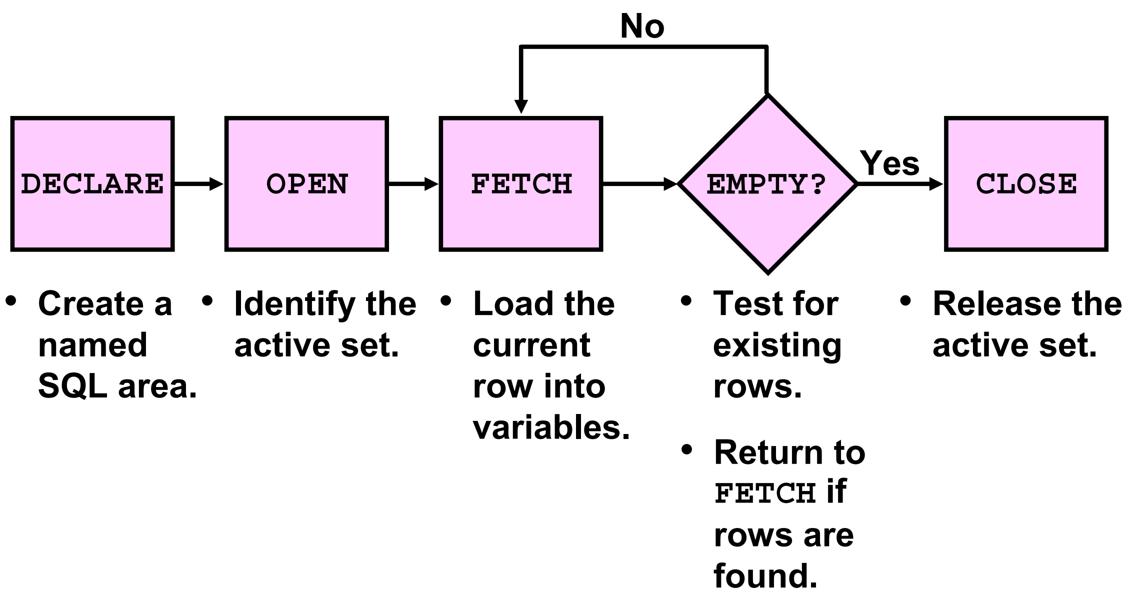
You declare explicit cursors in PL/SQL when you have a `SELECT` statement that returns multiple rows. You can process each row returned by the `SELECT` statement.

The set of rows returned by a multiple-row query is called the *active set*. Its size is the number of rows that meet your search criteria. The diagram in the slide shows how an explicit cursor "points" to the current row in the active set. This enables your program to process the rows one at a time.

Explicit cursor functions:

- Can do row-by-row processing beyond the first row returned by a query
- Keep track of which row is currently being processed
- Enable the programmer to manually control explicit cursors in the PL/SQL block

Controlling Explicit Cursors

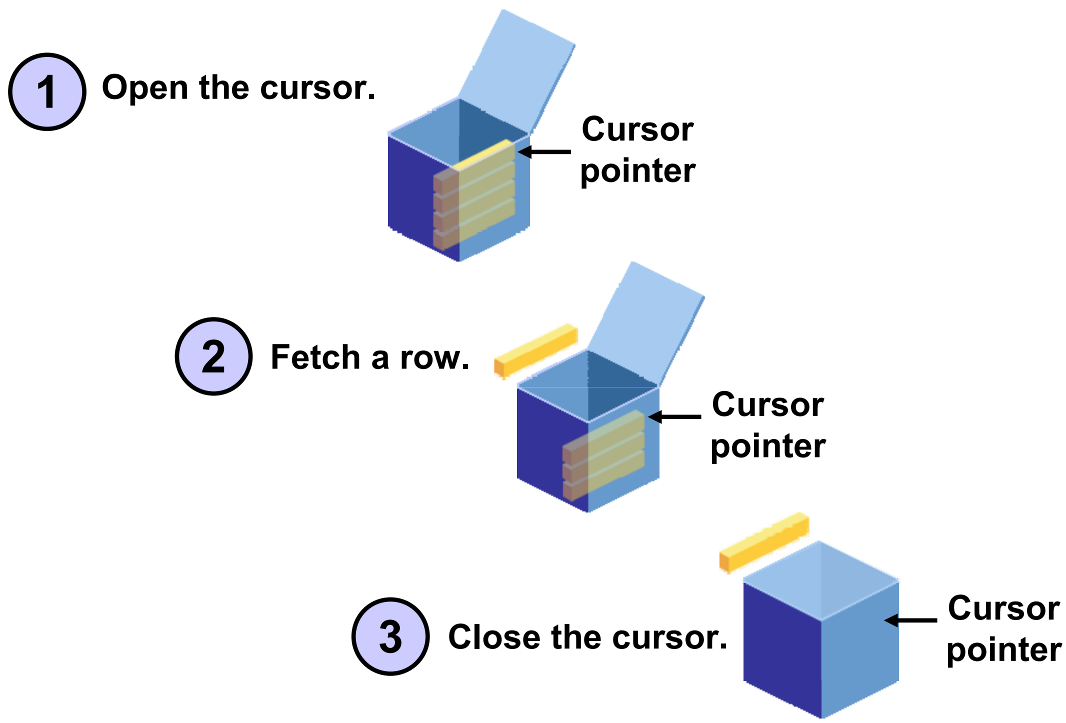


Controlling Explicit Cursors

Now that you have a conceptual understanding of cursors, review the steps to use them.

1. In the declarative section of a PL/SQL block, declare the cursor by naming it and defining the structure of the query to be associated with it.
2. Open the cursor.
   The `OPEN` statement executes the query and binds any variables that are referenced. Rows identified by the query are called the *active set* and are now available for fetching.
3. Fetch data from the cursor.
   In the flow diagram shown in the slide, after each fetch you test the cursor for any existing row. If there are no more rows to process, you must close the cursor.
4. Close the cursor.
   The `CLOSE` statement releases the active set of rows. It is now possible to reopen the cursor to establish a fresh active set.

Controlling Explicit Cursors



Controlling Explicit Cursors (continued)

A PL/SQL program opens a cursor, processes rows returned by a query, and then closes the cursor. The cursor marks the current position in the active set.

1. The `OPEN` statement executes the query associated with the cursor, identifies the active set, and positions the cursor at the first row.
2. The `FETCH` statement retrieves the current row and advances the cursor to the next row until either there are no more rows or until a specified condition is met.
3. The `CLOSE` statement releases the cursor.

Declaring the Cursor

Syntax:

```
CURSOR cursor_name IS
     select_statement;
```

Examples

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

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


Declaring the Cursor

The syntax to declare a cursor is shown in the slide. In the syntax:

cursor_name	Is a PL/SQL identifier
select_statement	Is a SELECT statement without an INTO clause

The active set of a cursor is determined by the SELECT statement in the cursor declaration. It is mandatory to have an INTO clause for a SELECT statement in PL/SQL. However, note that the SELECT statement in the cursor declaration cannot have an INTO clause. That is because you are only defining a cursor in the declarative section and not retrieving any rows into the cursor.

Note

- Do not include the INTO clause in the cursor declaration because it appears later in the FETCH statement.
- If processing rows in a specific sequence is required, use the ORDER BY clause in the query.
- The cursor can be any valid ANSI SELECT statement, including joins, subqueries, and so on.

Declaring the Cursor (continued)

The `emp_cursor` cursor is declared to retrieve the `employee_id` and `last_name` columns for those employees working in the department with a `department_id` of 30.

The `dept_cursor` cursor is declared to retrieve all the details for the department with the `location_id` 1700. Note that a variable is used while declaring the cursor. These variables are considered bind variables, which must be visible when you are declaring the cursor. These variables are examined only once at the time the cursor opens. You have learned that explicit cursors are used when you have to retrieve and operate on multiple rows in PL/SQL. However, this example shows that you can use the explicit cursor even if your `SELECT` statement returns only one row.

Opening the Cursor

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


Opening the Cursor

The OPEN statement executes the query associated with the cursor, identifies the active set, and positions the cursor pointer at the first row. The OPEN statement is included in the executable section of the PL/SQL block.

OPEN is an executable statement that performs the following operations:

1. Dynamically allocates memory for a context area
2. Parses the SELECT statement
3. Binds the input variables (sets the values for the input variables by obtaining their memory addresses)
4. Identifies the active set (the set of rows that satisfy the search criteria). Rows in the active set are not retrieved into variables when the OPEN statement is executed. Rather, the FETCH statement retrieves the rows from the cursor to the variables.
5. Positions the pointer to the first row in the active set

Note: If the query returns no rows when the cursor is opened, PL/SQL does not raise an exception. However, you can test the status of the implicit cursor after a fetch by using the SQL%ROWCOUNT cursor attribute. For explicit cursors, use <*cursor_name*>%ROWCOUNT.

Fetching Data from the Cursor

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


Fetching Data from the Cursor

The `FETCH` statement retrieves the rows from the cursor one at a time. After each fetch, the cursor advances to the next row in the active set. You can use the `%NOTFOUND` attribute to determine whether the entire active set has been retrieved.

Consider the example shown in the slide. Two variables, `empno` and `lname`, are declared to hold the fetched values from the cursor. Examine the `FETCH` statement.

The output of the PL/SQL block is as follows:

```
114 Raphaely
PL/SQL procedure successfully completed.
```

You have successfully fetched the values from the cursor to the variables. However, there are six employees in department 30, but only one row was fetched. To fetch all rows, you must use loops. In the next slide, you see how a loop is used to fetch all the rows.

The `FETCH` statement performs the following operations:

1. Reads the data for the current row into the output PL/SQL variables
2. Advances the pointer to the next row in the active set

Fetching Data from the Cursor (continued)

- Include the same number of variables in the `INTO` clause of the `FETCH` statement as there are columns in the `SELECT` statement, and be sure that the data types are compatible.
- Match each variable to correspond to the columns positionally.
- Alternatively, define a record for the cursor and reference the record in the `FETCH INTO` clause.
- Test to see whether the cursor contains rows. If a fetch acquires no values, there are no rows left to process in the active set and no error is recorded.

Fetching Data from the Cursor

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


Fetching Data from the Cursor (continued)

Observe that a simple LOOP is used to fetch all the rows. Also, the cursor attribute %NOTFOUND is used to test for the exit condition. The output of the PL/SQL block is:

```
114 Raphaely
115 Khoo
116 Baida
117 Tobias
118 Himuro
119 Colmenares
PL/SQL procedure successfully completed.
```

Closing the Cursor

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


Closing the Cursor

The CLOSE statement disables the cursor, releases the context area, and undefines the active set. Close the cursor after completing the processing of the FETCH statement. You can reopen the cursor if required. A cursor can be reopened only if it is closed. If you attempt to fetch data from a cursor after it has been closed, then an INVALID_CURSOR exception will be raised.

Note: Although it is possible to terminate the PL/SQL block without closing cursors, you should make it a habit to close any cursor that you declare explicitly to free up resources. There is a maximum limit on the number of open cursors per session, which is determined by the OPEN_CURSORS parameter in the database parameter file. (OPEN_CURSORS = 50 by default.)

Cursors and Records

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

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


Cursors and Records

You have already seen that you can define records that have the structure of columns in a table. You can also define a record based on the selected list of columns in an explicit cursor. This is convenient for processing the rows of the active set, because you can simply fetch into the record. Therefore, the values of the row are loaded directly into the corresponding fields of the record.

Cursor FOR Loops

Syntax:

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

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

Cursor FOR Loops

You have learned to fetch data from cursors by using simple loops. You now learn to use a cursor FOR loop, which processes rows in an explicit cursor. It is a shortcut because the cursor is opened, a row is fetched once for each iteration in the loop, the loop exits when the last row is processed, and the cursor is closed automatically. The loop itself is terminated automatically at the end of the iteration where the last row is fetched.

In the syntax:

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

Guidelines

- Do not declare the record that controls the loop; 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 FOR Loops

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


Cursor FOR Loops (continued)

The example that was used to demonstrate the usage of a simple loop to fetch data from cursors is rewritten to use the cursor FOR loop.

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

Explicit Cursor Attributes

Obtain status information about a cursor.

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

ORACLE

Explicit Cursor Attributes

As with implicit cursors, there are four attributes for obtaining status information about a cursor. When appended to the cursor variable name, these attributes return useful information about the execution of a cursor manipulation statement.

Note: You cannot reference cursor attributes directly in a SQL statement.

%ISOPEN Attribute

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

Example

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


%ISOPEN Attribute

- You can fetch rows only when the cursor is open. Use the `%ISOPEN` cursor attribute to determine whether the cursor is open.
- Fetch rows in a loop. Use cursor attributes to determine when to exit the loop.
- Use the `%ROWCOUNT` cursor attribute to do the following:
  - Process an exact number of rows
  - Fetch the rows in a loop and determine when to exit the loop

Note: `%ISOPEN` returns the status of the cursor: `TRUE` if open and `FALSE` if not.

%ROWCOUNT and %NOTFOUND: Example

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


%ROWCOUNT and %NOTFOUND: Example

The example in the slide retrieves the first ten employees one by one. This example shows how %ROWCOUNT and %NOTFOUND attributes can be used for exit conditions in a loop.

Cursor FOR Loops Using Subqueries

There is no need to declare the cursor.

Example

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


Cursor FOR Loops Using Subqueries

Note that there is no declarative section in this PL/SQL block. The difference between the cursor FOR loops using subqueries and the cursor FOR loop lies in the cursor declaration. If you are writing cursor FOR loops using subqueries, you need not declare the cursor in the declarative section. You have to provide the SELECT statement that determines the active set in the loop itself.

The example that was used to illustrate a cursor FOR loop is rewritten to illustrate a cursor FOR loop using subqueries.

Note: You cannot reference explicit cursor attributes if you use a subquery in a cursor FOR loop because you cannot give the cursor an explicit name.

Cursors with Parameters

Syntax:

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

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

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


Cursors with Parameters

You can pass parameters to a cursor in a cursor `FOR` loop. This means that you can open and close an explicit cursor several times in a block, returning a different active set on each occasion. For each execution, the previous cursor is closed and reopened with a new set of parameters.

Each formal parameter in the cursor declaration must have a corresponding actual parameter in the `OPEN` statement. Parameter data types are the same as those for scalar variables, but you do not give them sizes. The parameter names are for references in the query expression of the cursor.

In the syntax:

cursor_name	Is a PL/SQL identifier for the declared cursor
parameter_name	Is the name of a parameter
datatype	Is the scalar data type of the parameter
select_statement	Is a `SELECT` statement without the `INTO` clause

The parameter notation does not offer greater functionality; it simply allows you to specify input values easily and clearly. This is particularly useful when the same cursor is referenced repeatedly.

Cursors with Parameters

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


Cursors with Parameters (continued)

Parameter data types are the same as those for scalar variables, but you do not give them sizes. The parameter names are for reference in the cursor's query. In the following example, a cursor is declared and is defined with one parameter:

```
DECLARE
   CURSOR emp_cursor(deptno NUMBER) IS SELECT ...
```

The following statements open the cursor and return different active sets:

```
OPEN emp_cursor(10);
OPEN emp_cursor(20);
```

You can pass parameters to the cursor that is used in a cursor FOR loop:

```
DECLARE
CURSOR emp_cursor(p_deptno NUMBER, p_job VARCHAR2)IS
   SELECT ...
BEGIN
   FOR emp_record IN emp_cursor(10, 'Sales') LOOP ...
```

FOR UPDATE Clause

Syntax:

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

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

ORACLE

FOR UPDATE Clause

If there are multiple sessions for a single database, there is the possibility that the rows of a particular table were updated after you opened your cursor. You see the updated data only when you reopen the cursor. Therefore, it is better to have locks on the rows before you update or delete rows. You can lock the rows with the `FOR UPDATE` clause in the cursor query.

In the syntax:

column_reference	Is a column in the table against which the query is performed (A list of columns may also be used.)
`NOWAIT`	Returns an Oracle server error if the rows are locked by another session.

The `FOR UPDATE` clause is the last clause in a select statement, even after `ORDER BY` (if it exists). When querying multiple tables, you can use the `FOR UPDATE` clause to confine row locking to particular tables. `FOR UPDATE OF col_name(s)` locks rows only in tables that contain `col_name(s)`.

The `FOR UPDATE` Clause (continued)

The `SELECT ... FOR UPDATE` statement identifies the rows that are to be updated or deleted, and then locks each row in the result set. This is useful when you want to base an update on the existing values in a row. In that case, you must make sure that the row is not changed by another session before the update.

The optional `NOWAIT` keyword tells the Oracle server not to wait if requested rows have been locked by another user. Control is immediately returned to your program so that it can do other work before trying again to acquire the lock. If you omit the `NOWAIT` keyword, the Oracle server waits until the rows are available.

Example:

```
DECLARE
  CURSOR emp_cursor IS
  SELECT employee_id, last_name, FROM employees
  WHERE department_id = 80 FOR UPDATE OF salary NOWAIT;
  ...
```

If the Oracle server cannot acquire the locks on the rows it needs in a `SELECT FOR UPDATE`, it waits indefinitely. Use `NOWAIT` to handle such situations. If the rows are locked by another session and you have specified `NOWAIT`, opening the cursor results in an error. You can try to open the cursor later. You can use `WAIT` instead of `NOWAIT`, specify the number of seconds to wait, and determine whether the rows are unlocked. If the rows are still locked after *n* seconds, an error is returned.

It is not mandatory for the `FOR UPDATE OF` clause to refer to a column, but it is recommended for better readability and maintenance.

`WHERE CURRENT OF` Clause

Syntax:

```
WHERE CURRENT OF cursor ;
```

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

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


`WHERE CURRENT OF` Clause

The `WHERE CURRENT OF` clause is used in conjunction with the `FOR UPDATE` clause to refer to the current row in an explicit cursor. The `WHERE CURRENT OF` clause is used in the `UPDATE` or `DELETE` statement, whereas the `FOR UPDATE` clause is specified in the cursor declaration. You can use the combination for updating and deleting the current row from the corresponding database table. This enables you to apply updates and deletes to the row currently being addressed, without the need to explicitly reference the row ID. You must include the `FOR UPDATE` clause in the cursor query so that the rows are locked on `OPEN`.

In the syntax:

cursor	Is the name of a declared cursor (The cursor must have been declared with the `FOR UPDATE` clause.)

Cursors with Subqueries

Example

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


Cursors with Subqueries

A subquery is a query (usually enclosed by parentheses) that appears within another SQL statement. When evaluated, the subquery provides a value or set of values to the outer query. Subqueries are often used in the `WHERE` clause of a select statement. They can also be used in the `FROM` clause, creating a temporary data source for that query.

In the example in the slide, the subquery creates a data source consisting of department numbers and the number of employees in each department (known by the alias `STAFF`). A table alias, `t2`, refers to this temporary data source in the `FROM` clause. When this cursor is opened, the active set contains the department number, department name, and total number of employees working for those departments that have three or more employees.

Summary

In this lesson, you should have learned how to:

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

ORACLE

Summary

The Oracle server uses work areas to execute SQL statements and store processing information. You can use a PL/SQL construct called a *cursor* to name a work area and access its stored information. There are two kinds of cursors: implicit and explicit. PL/SQL implicitly declares a cursor for all SQL data manipulation statements, including queries that return only one row. For queries that return more than one row, you must explicitly declare a cursor to process the rows individually.

Every explicit cursor and cursor variable has four attributes: `%FOUND`, `%ISOPEN` `%NOTFOUND`, and `%ROWCOUNT`. When appended to the cursor variable name, these attributes return useful information about the execution of a SQL statement. You can use cursor attributes in procedural statements but not in SQL statements.

Use simple loops or cursor `FOR` loops to operate on the multiple rows fetched by the cursor. If you are using simple loops, you have to open, fetch, and close the cursor; however, cursor `FOR` loops do this implicitly. If you are updating or deleting rows, lock the rows by using a `FOR UPDATE` clause. This ensures that the data you are using is not updated by another session after you open the cursor. Use a `WHERE CURRENT OF` clause in conjunction with the `FOR UPDATE` clause to reference the current row fetched by the cursor.

Practice 7: Overview

This practice covers the following topics:

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

Practice 7: Overview

In this practice, you apply your knowledge of cursors to process a number of rows from a table and populate another table with the results using a cursor `FOR` loop. You also write a cursor with parameters.

Practice 7

1. Create a PL/SQL block that determines the top *n* salaries of the employees.
   a. Execute the script `lab_07_01.sql` to create a new table, `top_salaries`, for storing the salaries of the employees.
   b. Accept a number *n* from the user where *n* represents the number of top *n* earners from the `employees` table. For example, to view the top five salaries, enter 5.

      Note: Use the `DEFINE` command to define a variable `p_num` to provide the value for *n*. Pass the value to the PL/SQL block through an *i*SQL*Plus substitution variable.
   c. In the declarative section, declare two variables: `num` of type `NUMBER` to accept the substitution variable `p_num`, `sal` of type `employees.salary`. Declare a cursor, `emp_cursor`, that retrieves the salaries of employees in descending order. Remember that the salaries should not be duplicated.
   d. In the executable section, open the loop and fetch top *n* salaries and insert them into `top_salaries` table. You can use a simple loop to operate on the data. Also, try and use `%ROWCOUNT` and `%FOUND` attributes for the exit condition.
   e. After inserting into the `top_salaries` table, display the rows with a `SELECT` statement. The output shown represents the five highest salaries in the `employees` table.

SALARY
24000
17000
14000
13500
13000

   f. Test a variety of special cases, such as *n* = 0 or where *n* is greater than the number of employees in the `employees` table. Empty the `top_salaries` table after each test.
2. Create a PL/SQL block that does the following:
   a. Use the `DEFINE` command to define a variable `p_deptno` to provide the department ID.
   b. In the declarative section, declare a variable `deptno` of type `NUMBER` and assign the value of `p_deptno`.
   c. Declare a cursor, `emp_cursor`, that retrieves the `last_name`, `salary`, and `manager_id` of the employees working in the department specified in `deptno`.

Practice 7 (continued)

d. In the executable section use the cursor `FOR` loop to operate on the data retrieved. If the salary of the employee is less than 5000 and if the manager ID is either 101 or 124, display the message <<*last_name*>> Due for a raise. Otherwise, display the message <<*last_name*>> Not due for a raise.

e. Test the PL/SQL block for the following cases:

Department ID	Message
10	Whalen Due for a raise
20	Hartstein Not Due for a raise Fay Not Due for a raise
50	Weiss Not Due for a raise Fripp Not Due for a raise Kaufling Not Due for a raise Vollman Not Due for a raise Mourgas Not Due for a raise . . . . . . Rajs Due for a raise
80	Russel Not Due for a raise Partners Not Due for a raise Errazuriz Not Due for a raise Cambrault Not Due for a raise . . . . . .

Practice 7 (continued)

3. Write a PL/SQL block, which declares and uses cursors with parameters.
   In a loop, use a cursor to retrieve the department number and the department name from the `departments` table for a department whose `department_id` is less than 100. Pass the department number to another cursor as a parameter to retrieve from the `employees` table the details of employee last name, job, hire date, and salary of those employees whose `employee_id` is less than 120 and who work in that department.
   a. In the declarative section, declare a cursor `dept_cursor` to retrieve `department_id`, `department_name` for those departments with `department_id` less than 100. Order by `department_id`.
   b. Declare another cursor `emp_cursor` that takes the department number as parameter and retrieves `last_name`, `job_id`, `hire_date`, and `salary` of those employees with `employee_id` of less than 120 and who work in that department.
   c. Declare variables to hold the values retrieved from each cursor. Use the `%TYPE` attribute while declaring variables.
   d. Open the `dept_cursor`, use a simple loop and fetch values into the variables declared. Display the department number and department name.
   e. For each department, open the `emp_cursor` by passing the current department number as a parameter. Start another loop and fetch the values of `emp_cursor` into variables and print all the details retrieved from the `employees` table.

      Note: You may want to print a line after you have displayed the details of each department. Use appropriate attributes for the exit condition. Also determine whether a cursor is already open before opening the cursor.
   f. Close all the loops and cursors, and end the executable section. Execute the script.

Practice 7 (continued)

The sample output is shown below.

```
Department Number : 10 Department Name : Administration
-------------------------------------------------------------------------------
Department Number : 20 Department Name : Marketing
-------------------------------------------------------------------------------
Department Number : 30 Department Name : Purchasing
Raphaely PU_MAN 07-DEC-94 11000
Khoo PU_CLERK 18-MAY-95 3100
Baida PU_CLERK 24-DEC-97 2900
Tobias PU_CLERK 24-JUL-97 2800
Himuro PU_CLERK 15-NOV-98 2600
Colmenares PU_CLERK 10-AUG-99 2500
-------------------------------------------------------------------------------
Department Number : 40 Department Name : Human Resources
-------------------------------------------------------------------------------
Department Number : 50 Department Name : Shipping
-------------------------------------------------------------------------------
Department Number : 60 Department Name : IT
Hunold IT_PROG 03-JAN-90 9000
Ernst IT_PROG 21-MAY-91 6000
Austin IT_PROG 25-JUN-97 4800
Pataballa IT_PROG 05-FEB-98 4800
Lorentz IT_PROG 07-FEB-99 4200
-------------------------------------------------------------------------------
Department Number : 70 Department Name : Public Relations
-------------------------------------------------------------------------------
Department Number : 80 Department Name : Sales
-------------------------------------------------------------------------------
Department Number : 90 Department Name : Executive
King AD_PRES 17-JUN-87 24000
Kochhar AD_VP 21-SEP-89 17000
De Haan AD_VP 13-JAN-93 17000
-------------------------------------------------------------------------------
PL/SQL procedure successfully completed.
```

Practice 7 (continued)

4. Load the script lab_06_04_soln.sql.
   a. Look for the comment "DECLARE A CURSOR CALLED emp_records TO HOLD salary, first_name, and last_name of employees" and include the declaration. Create the cursor such that it retrieves the salary, first_name, and last_name of employees in the department specified by the user (substitution variable emp_deptid). Use the FOR UPDATE clause.
   b. Look for the comment "INCLUDE EXECUTABLE SECTION OF INNER BLOCK HERE" and start the executable block.
   c. Only employees working in the departments with department_id 20, 60, 80,100, and 110 are eligible for raises this quarter. Check if the user has entered any of these department IDs. If the value does not match, display the message "SORRY, NO SALARY REVISIONS FOR EMPLOYEES IN THIS DEPARTMENT." If the value matches, then, open the cursor emp_records.
   d. Start a simple loop and fetch the values into emp_sal, emp_fname, and emp_lname. Use %NOTFOUND for the exit condition.
   e. Include a CASE expression. Use the following table as reference for the conditions in the WHEN clause of the CASE expression.
   Note: In your CASE expression use the constants such as c_range1, c_hike1 which are already declared.

salary	**Hike percentage**
< 6500	20
> 6500 < 9500	15
> 9500 <12000	8
>12000	3

   For example, if the salary of the employee is less than 6500, then increase the salary by 20 percent. In every WHEN clause, concatenate the first_name and last_name of the employee and store it in the INDEX BY table. Increment the value in variable i so that you can store the string in the next location. Include an UPDATE statement with the WHERE CURRENT OF clause.
   f. Close the loop. Use the %ROWCOUNT attribute and print the number of records that were modified. Close the cursor.
   g. Include a simple loop to print the names of all the employees whose salaries were revised.
   Note: You already have the names of these employees in the INDEX BY table. Look for the comment "CLOSE THE INNER BLOCK" and include an END IF statement and an END statement.
   f. Save your script as lab_07_04_soln.sql.

8

Handling Exceptions

Objectives

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

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

Lesson Aim

You have learned to write PL/SQL blocks with a declarative section and an executable section. All the SQL and PL/SQL code that must be executed is written in the executable block.

So far we have assumed that the code works satisfactorily if we take care of compile-time errors. However, the code may cause some unanticipated errors at run time. In this lesson, you learn how to deal with such errors in the PL/SQL block.

Example of an Exception

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


Example of an Exception

Consider the example shown in the slide. There are no syntax errors in the code, which means you must be able to successfully execute the anonymous block. The select statement in the block retrieves the `last_name` of John. You see the following output when you execute the code:

```
DECLARE
*

ERROR at line 1:
ORA-01422: exact fetch returns more than requested number of rows
ORA-06512: at line 4
```

The code does not work as expected. You expected the `SELECT` statement to retrieve only one row; however, it retrieves multiple rows. Such errors that occur at run time are called *exceptions*. When an exception occurs, the PL/SQL block is terminated. You can handle such exceptions in your PL/SQL block.

Example of an Exception

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


Example of an Exception (continued)

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, you 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 EXCEPTION section of the code in the slide. You need not pay attention to the syntax and statements; you learn about them later in the lesson.

The code in the previous slide is rewritten to handle the exception that occurred. The output of the code is:

```
Your select statement retrieved multiple rows. Consider using a cursor.
PL/SQL procedure successfully completed.
```

Unlike earlier, the PL/SQL program does not terminate abruptly. When the exception is raised, the control shifts to the exception section and all the statements in the exception section are executed. The PL/SQL block terminates with normal, successful completion.

Handling Exceptions with PL/SQL

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

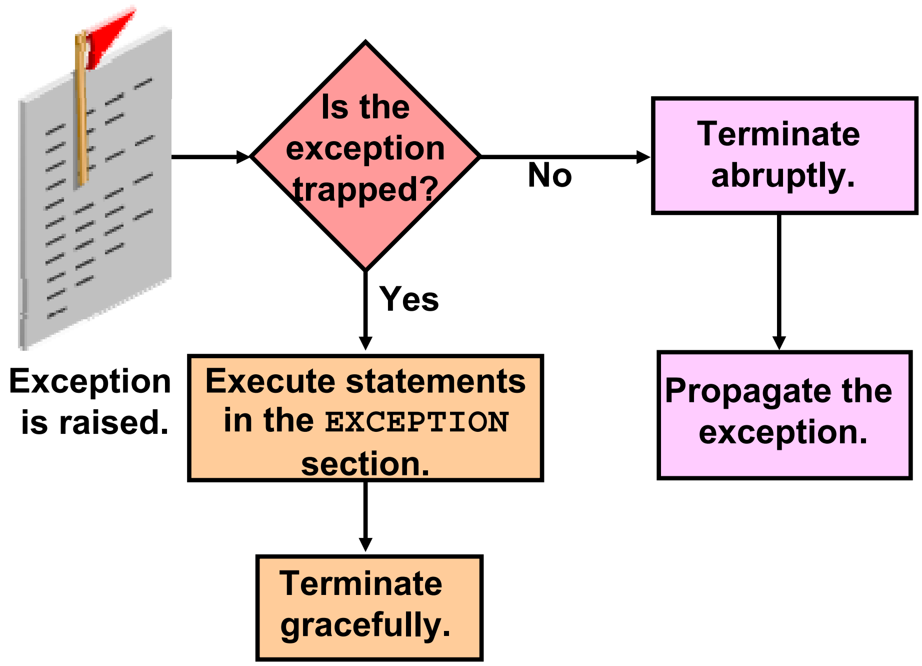
Handling Exceptions with PL/SQL

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.

Two 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 exception `NO_DATA_FOUND`. These errors are converted into predefined exceptions.
- Depending on the business functionality your program implements, you may have to explicitly raise an exception. You raise an exception explicitly by issuing the `RAISE` statement in the block. The raised exception may be either user-defined or predefined. There are also some non-predefined Oracle errors. These errors are any standard Oracle errors that are not predefined. You can explicitly declare exceptions and associate them with the non-predefined Oracle errors.

Handling Exceptions



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 then 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).

Exception Types

- Predefined Oracle server
- Non-predefined Oracle server

$\}$ Implicitly raised

- User-defined

Explicitly raised

Exception Types

There are three types of exceptions.

Exception	Description	Directions for Handling
Predefined Oracle Server error	One of approximately 20 errors that occur most often in PL/SQL code	You need not declare these exceptions. They are predefined by the Oracle server and are raised implicitly.
Non-predefined Oracle Server error	Any other standard Oracle Server error	Declare within the declarative section and enable the Oracle server to raise them implicitly.
User-defined error	A condition that the developer determines is abnormal	Declare in the declarative section and raise explicitly.

Note: Some application tools with client-side PL/SQL (such as Oracle Developer Forms) have their own exceptions.

Trapping Exceptions

Syntax:

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


Trapping Exceptions

You can trap any error by including a corresponding handler within the exception handling section of the PL/SQL block. Each handler consists of a `WHEN` clause, which specifies an exception name, followed by a sequence of statements to be executed when that exception is raised. You can include any number of handlers within an `EXCEPTION` section to handle specific exceptions. However, you cannot have multiple handlers for a single exception.

In the syntax:

exception	Is the standard name of a predefined exception or the name of a user-defined exception declared within the declarative section
statement	Is one or more PL/SQL or SQL statements
`OTHERS`	Is an optional exception-handling clause that traps any exceptions that have not been explicitly handled

Trapping Exceptions (continued)

`WHEN OTHERS` Exception Handler

The exception-handling section traps only those exceptions that are specified; any other exceptions are not trapped unless you use the `OTHERS` exception handler. This traps any exception not yet handled. For this reason, `OTHERS` may be used, and if used it must be the last exception handler that is defined.

```
WHEN NO_DATA_FOUND THEN
 statement1;
 ...
WHEN TOO_MANY_ROWS THEN
 statement1;
 ...
WHEN OTHERS THEN
 statement1;
```

Consider the preceding example. If the exception `NO_DATA_FOUND` is raised by the program, the statements in the corresponding handler are executed. If the exception `TOO_MANY_ROWS` is raised, the statements in the corresponding handler are executed. However, if some other exception is raised, the statements in the `OTHERS` exception handler are executed.

The `OTHERS` handler traps all the exceptions that are not already trapped. Some Oracle tools have their own predefined exceptions that you can raise to cause events in the application. The `OTHERS` handler also traps these exceptions.

Guidelines for Trapping Exceptions

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

Guidelines for Trapping Exceptions

- Begin the exception-handling section of the block with the `EXCEPTION` keyword.
- Define several exception handlers, each with its own set of actions, for the block.
- When an exception occurs, PL/SQL processes only one handler before leaving the block.
- Place the `OTHERS` clause after all other exception-handling clauses.
- You can have only one `OTHERS` clause.
- Exceptions cannot appear in assignment statements or SQL statements.

Trapping Predefined Oracle Server Errors

- **Reference the predefined name in the exception-handling routine.**
- **Sample predefined exceptions:**
  - `NO_DATA_FOUND`
  - `TOO_MANY_ROWS`
  - `INVALID_CURSOR`
  - `ZERO_DIVIDE`
  - `DUP_VAL_ON_INDEX`

Trapping Predefined Oracle Server Errors

Trap a predefined Oracle server error by referencing 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.

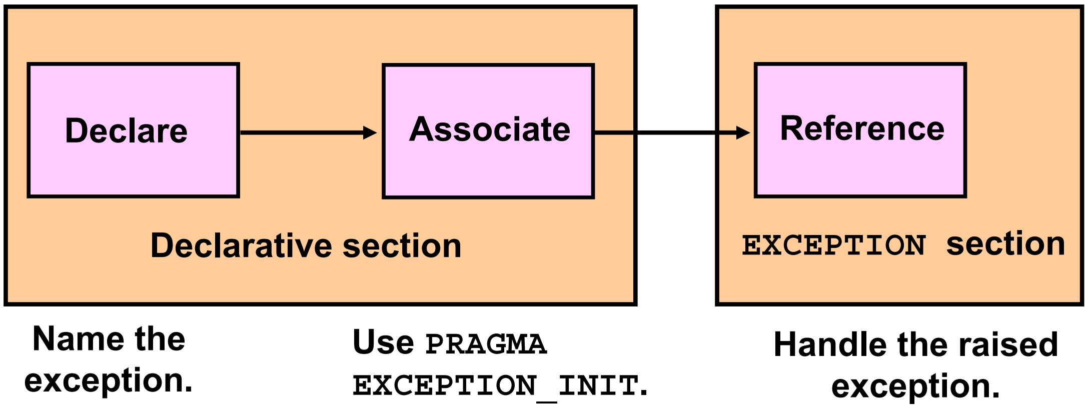
Predefined Exceptions

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 are selected, and there is no ELSE clause.
COLLECTION_IS_NULL	ORA-06531	Attempted to apply collection methods other than EXISTS to an uninitialized nested table or VARRAY
CURSOR_ALREADY_OPEN	ORA-06511	Attempted to open an already-open cursor
DUP_VAL_ON_INDEX	ORA-00001	Attempted to insert a duplicate value
INVALID_CURSOR	ORA-01001	Illegal cursor operation occurred.
INVALID_NUMBER	ORA-01722	Conversion of character string to number fails.
LOGIN_DENIED	ORA-01017	Logging on to the Oracle server with an invalid username or password
NO_DATA_FOUND	ORA-01403	Single row SELECT returned no data.
NOT_LOGGED_ON	ORA-01012	PL/SQL program issues a database call without being connected to the Oracle server.
PROGRAM_ERROR	ORA-06501	PL/SQL has an internal problem.
ROWTYPE_MISMATCH	ORA-06504	Host cursor variable and PL/SQL cursor variable involved in an assignment have incompatible return types.

Predefined Exceptions (continued)

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

Trapping Non-Predefined Oracle Server Errors



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 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` tells the compiler to associate an exception name with an Oracle error number. That enables you to refer to any internal exception by name and to write a specific handler for it.

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

Non-Predefined Error

To trap Oracle server error number –01400 (“cannot insert `NULL`”):

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


Non-Predefined Error

1. Declare the name of the exception in the declarative section.
   Syntax:
   `exception    EXCEPTION;`
   In the syntax, *exception* is the name of the exception.
2. Associate the declared exception with the standard Oracle server error number using the `PRAGMA EXCEPTION_INIT` function.
   Syntax:
   `PRAGMA EXCEPTION_INIT(exception, error_number);`
   In the syntax, *exception* is the previously declared exception and *error_number* is a standard Oracle server error number.
3. Reference the declared exception within the corresponding exception-handling routine.

Example

The example in the slide tries to insert the value `NULL` for the `department_name` column of the `departments` table. However, the operation is not successful because `department_name` is a `NOT NULL` column. Note the following line in the example:

```
DBMS_OUTPUT.PUT_LINE(SQLERRM);
```

The `SQLERRM` function is used to retrieve the error message. You learn more about `SQLERRM` in the next few slides.

Functions for Trapping Exceptions

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

Functions for Trapping Exceptions

When an exception occurs, you can identify the associated error code or error message by using two functions. Based on the values of the code or the message, you can decide which subsequent actions to take.

`SQLCODE` returns the Oracle error number for internal exceptions. `SQLERRM` returns the message associated with the error number.

Function	Description
`SQLCODE`	Returns the numeric value for the error code (You can assign it to a `NUMBER` variable.)
`SQLERRM`	Returns character data containing the message associated with the error number

`SQLCODE` Values: Examples

SQLCODE Value	Description
0	No exception encountered
1	User-defined exception
+100	`NO_DATA_FOUND` exception
negative number	Another Oracle server error number

Functions for Trapping Exceptions

Example

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

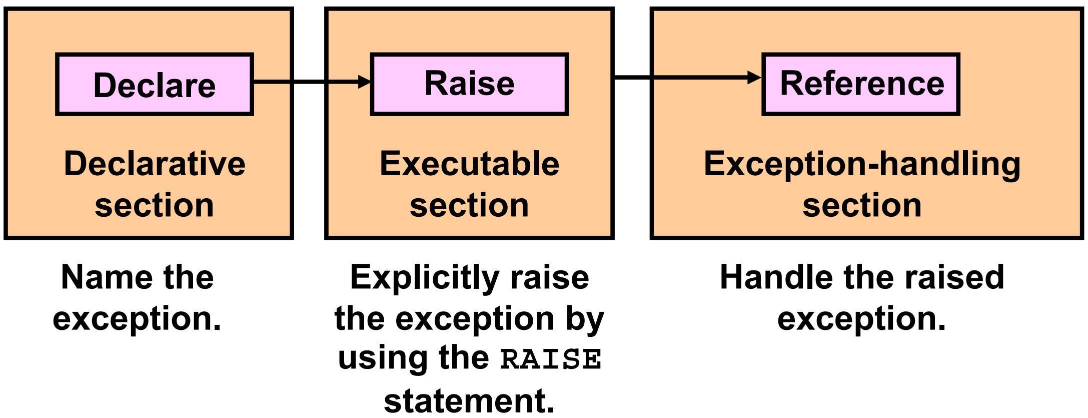

Functions for Trapping Exceptions (continued)

When an exception is trapped in the WHEN OTHERS exception handler, you can use a set of generic functions to identify those errors. The example in the slide illustrates the values of SQLCODE and SQLERRM assigned to variables, and then those variables being used in a SQL statement.

You cannot use SQLCODE or SQLERRM directly in a SQL statement. Instead, you must assign their values to local variables and then use the variables in the SQL statement, as shown in the following example:

```
DECLARE
   err_num NUMBER;
   err_msg VARCHAR2(100);
BEGIN
   ...
EXCEPTION
   ...
   WHEN OTHERS THEN
      err_num := SQLCODE;
      err_msg := SUBSTR(SQLERRM, 1, 100);
      INSERT INTO errors VALUES (err_num, err_msg);
END;
/
```

Trapping User-Defined Exceptions



Trapping User-Defined Exceptions

PL/SQL enables you to define your own exceptions depending on the requirements of your application. For example, you may prompt the user to enter a department number. Define an exception to deal with error conditions in the input data. Check whether the department number exists. If it does not, then you may have to raise the user-defined exception.

PL/SQL exceptions must be:

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

Trapping User-Defined Exceptions

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

1

2

3

Trapping User-Defined Exceptions (continued)

You trap a user-defined exception by declaring it and raising it explicitly.

1. Declare the name of the user-defined exception within the declarative section.
   Syntax:
   `exception EXCEPTION;`
   In the syntax, *exception* is the name of the exception.
2. Use the `RAISE` statement to raise the exception explicitly within the executable section.
   Syntax:
   `RAISE exception;`
   In the syntax, *exception* is the previously declared exception.
3. Reference the declared exception within the corresponding exception-handling routine.

Example

This block updates the `department_name` of a department. The user supplies the department number and the new name. If the user enters a department number that does not exist, no rows are updated in the `departments` table. Raise an exception and print a message for the user that an invalid department number was entered.

Note: Use the `RAISE` statement by itself within an exception handler to raise the same exception again and propagate it back to the calling environment.

Calling Environments

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

Calling Environments

Instead of trapping an exception within the PL/SQL block, propagate the exception to allow the calling environment to handle it. Each calling environment has its own way of displaying and accessing errors.

Propagating Exceptions in a Subblock

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

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

ORACLE

Propagating Exceptions in a Subblock

When a subblock handles an exception, it terminates normally. Control resumes in the enclosing block immediately after the subblock's `END` statement.

However, if a PL/SQL raises an exception and the current block does not have a handler for that exception, the exception propagates to successive enclosing blocks until it finds a handler. If none of these blocks handle the exception, an unhandled exception in the host environment results.

When the exception propagates to an enclosing block, the remaining executable actions in that block are bypassed.

One advantage of this behavior is that you can enclose statements that require their own exclusive error handling in their own block, while leaving more general exception handling to the enclosing block.

Note in the example that the exceptions, `no_rows` and `integrity`, are declared in the outer block. In the inner block, when the `no_rows` exception is raised, PL/SQL looks for the exception to be handled in the subblock. Because the exception is not handled in the subblock, the exception propagates to the outer block, where PL/SQL finds the handler.

RAISE_APPLICATION_ERROR Procedure

Syntax:

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

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

RAISE_APPLICATION_ERROR Procedure

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 –20000 and –20999
message	Is the user-specified message for the exception; 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, which is the default, the error replaces all previous errors.)

RAISE_APPLICATION_ERROR Procedure

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

RAISE_APPLICATION_ERROR Procedure (continued)

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 produces a predefined, non-predefined, or user-defined error. The error number and message are displayed to the user.

RAISE_APPLICATION_ERROR Procedure

Executable section:

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

Exception section:

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


RAISE_APPLICATION_ERROR Procedure (continued)

The slide shows that the RAISE_APPLICATION_ERROR procedure can be used in both the executable and the exception sections of a PL/SQL program.

Here is another example of using the RAISE_APPLICATION_ERROR procedure:

```
DECLARE
  e_name EXCEPTION;
  PRAGMA EXCEPTION_INIT (e_name, -20999);
BEGIN
  ...
  DELETE FROM employees
  WHERE  last_name = 'Higgins';
  IF SQL%NOTFOUND THEN
   RAISE_APPLICATION_ERROR(-20999,'This is not a
   valid last name');
  END IF;
EXCEPTION
  WHEN e_name THEN
  -- handle the error
  ...
END;
/
```

Summary

In this lesson, you should have learned how to:

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

Summary

In this lesson, you learned how to deal with different types of exceptions. In PL/SQL, a warning or error condition at run time is called an exception. Predefined exceptions are error conditions that are defined by the Oracle server. Non-predefined exceptions can be any standard Oracle server errors. User-defined exceptions are exceptions specific to your application. The `PRAGMA EXCEPTION_INIT` function can be used to associate a declared exception name with an Oracle server error.

You can define exceptions of your own in the declarative section of any PL/SQL block. For example, you can define an exception named `INSUFFICIENT_FUNDS` to flag overdrawn bank accounts.

When an error occurs, an exception is raised. Normal execution stops and transfers control to the exception-handling section of your PL/SQL block. Internal exceptions are raised implicitly (automatically) by the run-time system; however, user-defined exceptions must be raised explicitly. To handle raised exceptions, you write separate routines called exception handlers.

Practice 8: Overview

This practice covers the following topics:

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

Practice 8: Overview

In this practice, you create exception handlers for specific situations.

Practice 8

1. The purpose of this example is to show the usage of predefined exceptions. Write a PL/SQL block to select the name of the employee with a given salary value.
   a. Delete all records in the `messages` table. Use the `DEFINE` command to define a variable `sal` and initialize it to 6000.
   b. In the declarative section declare two variables: `ename` of type `employees.last_name` and `emp_sal` of type `employees.salary`. Pass the value of the substitution variables to `emp_sal`.
   c. In the executable section, retrieve the last names of employees whose salaries are equal to the value in `emp_sal`.
   Note: Do not use explicit cursors.
   If the salary entered returns only one row, insert into the `messages` table the employee's name and the salary amount.
   d. If the salary entered does not return any rows, handle the exception with an appropriate exception handler and insert into the `messages` table the message "No employee with a salary of <*salary*>."
   e. If the salary entered returns more than one row, handle the exception with an appropriate exception handler and insert into the `messages` table the message "More than one employee with a salary of <*salary*>."
   f. Handle any other exception with an appropriate exception handler and insert into the `messages` table the message "Some other error occurred."
   g. Display the rows from the `messages` table to check whether the PL/SQL block has executed successfully. Sample output is shown below.

RESULTS
More than one employee with a salary of 6000

2. The purpose of this example is to show how to declare exceptions with a standard Oracle server error. Use the Oracle server error ORA-02292 (integrity constraint violated – child record found).
   a. In the declarative section, declare an exception `childrecord_exists`. Associate the declared exception with the standard Oracle server error –02292.
   b. In the executable section, display 'Deleting department 40.....'. Include a `DELETE` statement to delete the department with `department_id` 40.

c. Include an exception section to handle the `childrecord_exists` exception and display the appropriate message. Sample output is shown below.

```
Deleting department 40........
Cannot delete this department. There are employees in this department
(child records exist.)
PL/SQL procedure successfully completed.
```

3. Load the script `lab_07_04_soln.sql`.
   a. Observe the declarative section of the outer block. Note that the `no_such_employee` exception is declared.
   b. Look for the comment "RAISE EXCEPTION HERE." If the value of `emp_id` is not between 100 and 206, then raise the `no_such_employee` exception.
   c. Look for the comment "INCLUDE EXCEPTION SECTION FOR OUTER BLOCK" and handle the exceptions `no_such_employee` and `too_many_rows`. Display appropriate messages when the exceptions occur. The `employees` table has only one employee working in the `HR` department and therefore the code is written accordingly. The `too_many_rows` exception is handled to indicate that the select statement retrieves more than one employee working in the `HR` department.
   d. Close the outer block.
   e. Save your script as `lab_08_03_soln.sql`.
   f. Execute the script. Enter the employee number and the department number and observe the output. Enter different values and check for different conditions. The sample output for employee ID 203 and department ID 100 is shown below.

```
NUMBER OF RECORDS MODIFIED : 6
The following employees' salaries are updated
Nancy Greenberg
Daniel Faviet
John Chen
Ismael Sciarra
Jose Manuel Urman
Luis Popp
PL/SQL procedure successfully completed.
```

Creating Stored Procedures and Functions

Objectives

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

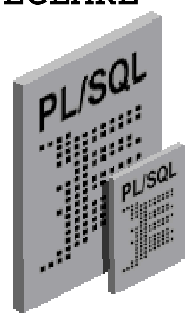
- **Differentiate between anonymous blocks and subprograms**
- **Create a simple procedure and invoke it from an anonymous block**
- **Create a simple function**
- **Create a simple function that accepts a parameter**
- **Differentiate between procedures and functions**

Lesson Aim

You have learned about anonymous blocks. This lesson introduces you to named blocks, which are also called *subprograms*. Procedures and functions are PL/SQL subprograms. In the lesson, you learn to differentiate between anonymous blocks and subprograms.

Procedures and Functions

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



Procedures and Functions

Until this point, anonymous blocks are the only examples of PL/SQL code covered in this course. As the name indicates, *anonymous* blocks are unnamed executable PL/SQL blocks. Because they are unnamed, they can be neither reused nor stored for later use.

Procedures and functions are named PL/SQL blocks. They are also known as subprograms. These subprograms are compiled and stored in the database. The block structure of the subprograms is similar to the structure of anonymous blocks. Subprograms can be declared not only at the schema level but also within any other PL/SQL block. A subprogram contains the following sections:

Declarative section: Subprograms can have an optional declarative section. However, unlike anonymous blocks, the declarative section of a subprogram does not start with the keyword `DECLARE`. The optional declarative section follows the keyword `IS` or `AS` in the subprogram declaration.

Executable section: This is the mandatory section of the subprogram, which contains the implementation of the business logic. Looking at the code in this section, you can easily determine the business functionality of the subprogram. This section begins and ends with the keywords `BEGIN` and `END`, respectively.

Exception section: This is an optional section that is included to handle exceptions.

Differences Between Anonymous Blocks and Subprograms

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

Differences Between Anonymous Blocks and Subprograms

The table in the slide not only shows the differences between anonymous blocks and subprograms, but also highlights the general benefits of subprograms.

Anonymous blocks are not persistent database objects. They are compiled and executed only once. They are not stored in the database for reuse. If you want to reuse, you must rerun the script that creates the anonymous block, which causes recompilation and execution. Procedures and functions are compiled and stored in the database in a compiled form. They are recompiled only when they are modified. Because they are stored in the database, any application can make use of these subprograms based on appropriate permissions. The calling application can pass parameters to the procedures if the procedure is designed to accept parameters. Similarly, a calling application can retrieve a value if it invokes a function or a procedure.

Procedure: Syntax

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


Procedure: Syntax

The slide shows the syntax for creating procedures. In the syntax:

procedure_name	Is the name of the procedure to be created
argument	Is the name given to the procedure parameter. Every argument is associated with a mode and data type. You can have any number of arguments separated by commas.

mode	Mode of argument:
	`IN` (default)
	`OUT`
	`IN OUT`

datatype	Is the data type of the associated parameter. The data type of parameters cannot have explicit size; instead, use `%TYPE`.
Procedure_body	Is the PL/SQL block that makes up the code

The argument list is optional in a procedure declaration. You learn about procedures in detail in the course titled *Oracle Database 10g: Develop PL/SQL Program Units*.

Procedure: Example

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


Procedure: Example

Examine the code in the slide. The `add_dept` procedure inserts a new department with department ID `280` and department name `ST-Curriculum`. The procedure declares two variables, `dept_id` and `dept_name`, in the declarative section. The declarative section of a procedure starts immediately after the procedure declaration and does not begin with the keyword `DECLARE`. The procedure uses the implicit cursor attribute or the `SQL%ROWCOUNT` SQL attribute to verify whether the row was successfully inserted. `SQL%ROWCOUNT` should return `1` in this case.

Note: When you create any object (such as a table, procedure, function, and so on), the entries are made to the `user_objects` table. When the code in the slide is executed successfully, you can check the `user_objects` table by issuing the following command:

```
SELECT object_name,object_type FROM user_objects;
```

ADD_DEPT	PROCEDURE
DEPT	TABLE

Procedure: Example (continued)

The source of the procedure is stored in the `user_source` table. You can check the source for the procedure by issuing the following command:

```
SELECT * FROM  user_source WHERE name='ADD_DEPT';
```

NAME	TYPE	LINE	TEXT
ADD_DEPT	PROCEDURE	1	PROCEDURE add_dept IS
ADD_DEPT	PROCEDURE	2	dept_id dept.department_id%TYPE;
ADD_DEPT	PROCEDURE	3	dept_name dept.department_name%TYPE;
ADD_DEPT	PROCEDURE	4	BEGIN
ADD_DEPT	PROCEDURE	5	dept_id:=280;
ADD_DEPT	PROCEDURE	6	dept_name:='ST-Curriculum';
ADD_DEPT	PROCEDURE	7	INSERT INTO dept(department_id,department_name)
ADD_DEPT	PROCEDURE	8	VALUES(dept_id,dept_name);
ADD_DEPT	PROCEDURE	9	DBMS_OUTPUT.PUT_LINE(' Inserted '|| SQL%ROWCOUNT ||' row ');
ADD_DEPT	PROCEDURE	10	END;
ADD_DEPT	PROCEDURE	11	

Invoking the Procedure

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

Inserted 1 row
PL/SQL procedure successfully completed.

DEPARTMENT_ID	DEPARTMENT_NAME
280	ST-Curriculum

Invoking the Procedure

The slide shows how to invoke a procedure from an anonymous block. You have to include the call to the procedure in the executable section of the anonymous block. Similarly, you can invoke the procedure from any application, such as a forms application, Java application and so on. The select statement in the code checks to see if the row was successfully inserted.

You can also invoke a procedure with the SQL statement `CALL <procedure_name>`.

Function: Syntax

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


Function: Syntax

The slide shows the syntax for creating a function. In the syntax:

function_name	Is the name of the function to be created
argument	Is the name given to the function parameter (Every argument is associated with a mode and data type. You can have any number or arguments separated by a comma. You pass the argument when you invoke the function.)
mode	Is the type of parameter (Only `IN` parameters should be declared.)
datatype	Is the data type of the associated parameter
`RETURN` *datatype*	Is the data type of the value returned by the function
function_body	Is the PL/SQL block that makes up the function code

The argument list is optional in function declaration. The difference between a procedure and a function is that a function must return a value to the calling program. Therefore, the syntax contains *return_type*, which specifies the data type of the value that the function returns. A procedure may return a value via an `OUT` or `IN OUT` parameter.

Function: Example

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


Function: Example

The `check_sal` function is written to determine whether the salary of a particular employee is greater than or less than the average salary of all employees working in the same department. The function returns `TRUE` if the salary of the employee is greater than the average salary of employees in the department; if not, it returns `FALSE`. The function returns `NULL` if a `NO_DATA_FOUND` exception is thrown.

Note that the function checks for the employee with the employee ID 205. The function is hard-coded to check for this employee ID only. If you want to check for any other employees, you must modify the function itself. You can solve this problem by declaring the function so that it accepts an argument. You can then pass the employee ID as parameter.

Invoking the Function

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

```
Salary > average
PL/SQL procedure successfully completed.
```


Invoking the Function

You include the call to the function in the executable section of the anonymous block. The function is invoked as a part of a statement. Remember that the `check_sal` function returns `Boolean` or `NULL`. Thus the call to the function is included as the conditional expression for the `IF` block.

Note: You can use the `DESCRIBE` command to check the arguments and return type of the function, as in the following example:

```
DESCRIBE check_sal;
```

Passing a Parameter to the Function

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

ORACLE

Passing a Parameter to the Function

Remember that the function was hard-coded to check the salary of the employee with the employee ID 205. The code shown in the slide removes that constraint because it is re-written to accept the employee number as a parameter. You can now pass different employee numbers and check for the employee's salary.

You learn more about functions in the course titled *Oracle Database 10g: Develop PL/SQL Program Units*.

Invoking the Function with a Parameter

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


Invoking the Function with a Parameter

The code in the slide invokes the function twice by passing parameters. The output of the code is as follows:

```
Checking for employee with id 205
Salary > average
Checking for employee with id 70
The function returned NULL due to exception
PL/SQL procedure successfully completed.
```

Summary

In this lesson, you should have learned how to:

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

Summary

You can use anonymous blocks to design any functionality in PL/SQL. However, the major constraint with anonymous blocks is that they are not stored and therefore cannot be reused.

Instead of creating anonymous blocks, you can create PL/SQL subprograms. Procedures and functions are called subprograms, which are named PL/SQL blocks. Subprograms express reusable logic by virtue of parameterization. The structure of a procedure or a function is similar to the structure of an anonymous block. These subprograms are stored in the database and are therefore reusable.

Practice 9: Overview

This practice covers the following topics:

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

Practice 9

1. In *i*SQL*Plus, load the script `lab_02_04_soln.sql` that you created for question 4 of practice 2.
   a. Modify the script to convert the anonymous block to a procedure called `greet`.
   b. Execute the script to create the procedure.
   c. Save your script as `lab_09_01_soln.sql`.
   d. Click the Clear button to clear the workspace.
   e. Create and execute an anonymous block to invoke the procedure `greet`. Sample output is shown below.

```
Hello World
TODAY IS : 20-JAN-04
TOMORROW IS : 21-JAN-04
PL/SQL procedure successfully completed.
```

2. Load the script `lab_09_01_soln.sql`.
   a. Drop the procedure `greet` by issuing the following command:
      `DROP PROCEDURE greet`
   b. Modify the procedure to accept an argument of type `VARCHAR2`. Call the argument `name`.
   c. Print Hello <*name*> instead of printing Hello World.
   d. Save your script as `lab_09_02_soln.sql`.
   e. Execute the script to create the procedure.
   f. Create and execute an anonymous block to invoke the procedure `greet` with a parameter. Sample output is shown below.

```
Hello Neema
TODAY IS : 20-JAN-04
TOMORROW IS : 21-JAN-04
PL/SQL procedure successfully completed.
```

A
Practice Solutions

Practice 1

Before you begin this practice, ensure that you have seen both the viewlets on *i*SQL*Plus usage.

The `labs` folder is the working directory where you can save your scripts. Ask your instructor for help in locating the `labs` folder for this course. The solutions for all practices are in the `soln` folder.

1. Which of the following PL/SQL blocks execute successfully?

   a.
   ```
   BEGIN
   END;
   ```

   b.
   ```
   DECLARE
   amount INTEGER(10);
   END;
   ```

   c.
   ```
   DECLARE
   BEGIN
   END;
   ```

   d.
   ```
   DECLARE
   amount INTEGER(10);
   BEGIN
   DBMS_OUTPUT.PUT_LINE(amount);
   END;
   ```

   *The block in **a** does not execute because the executable section does not have any statements. The block in **b** does not have the mandatory executable section that begins with the **`BEGIN`** keyword.*

   *The block in **c** has all the necessary parts but the executable section does not have any statements.*

2. Create and execute a simple anonymous block that outputs "Hello World." Execute and save this script as `lab_01_02_soln.sql`.

   a. Start *i*SQL*Plus. Provide login details. The instructor will provide the necessary information.

   b. Type the following code in the workspace.

```
SET SERVEROUTPUT ON
BEGIN
DBMS_OUTPUT.PUT_LINE(' Hello World ');
END;
```

   c. Click the Execute button.

d. You should see the following output:

```
Hello World
PL/SQL procedure successfully completed.
```

e. Click the Save Script button. Select the folder in which you want to save the file. Enter `lab_01_02_soln.sql` for the file name and click the Save button.

Practice 2

Note: Use *i*SQL*Plus for this practice.

1. Identify valid and invalid identifiers:

a.	`today`	**Valid**
b.	`last_name`	**Valid**
c.	`today's_date`	**Invalid** – character '’' is not allowed
d.	`Number_of_days_in_February_this_year`	**Invalid** – Too long
e.	`Isleap$year`	**Valid**
f.	`#number`	**Invalid** – Cannot start with '#'
g.	`NUMBER#`	**Valid**
h.	`number1to7`	**Valid**

2. Identify valid and invalid variable declaration and initialization:

a.	`number_of_copies`	`PLS_INTEGER;`	**Valid**
b.	`PRINTER_NAME`	constant `VARCHAR2(10);`	**Invalid**
c.	`deliver_to`	`VARCHAR2(10)`:=Johnson;	**Invalid**
d.	`by_when`	`DATE`:= `SYSDATE`+1;	**Valid**

*The declaration in **b** is invalid because constant variables must be initialized during declaration.*

*The declaration in **c** is invalid because string literals should be enclosed within single quotes.*

3. Examine the following anonymous block and choose the appropriate statement.

```
SET SERVEROUTPUT ON
DECLARE
fname VARCHAR2(20);
lname VARCHAR2(15) DEFAULT 'fernandez';
BEGIN
DBMS_OUTPUT.PUT_LINE( FNAME ||' ' ||lname);
END;
```

a. The block executes successfully and prints "fernandez."
b. The block produces an error because the `fname` variable is used without initializing.
c. The block executes successfully and prints "null fernandez."
d. The block produces an error because you cannot use the `DEFAULT` keyword to initialize a variable of type `VARCHAR2`.
e. The block produces an error because the `fname` variable is not declared.

a. The block will execute successfully and print "fernandez."

4. Create an anonymous block. In *i*SQL*Plus, load the script `lab_01_02_soln.sql`, which you created in exercise 2 of practice 1 by following these instructions:
   Click the Load Script button.
   Browse to select the `lab_01_02_soln.sql` file. Click the Load button. Your workspace will now have the code in the .sql file.

a. Add declarative section to this PL/SQL block. In the declarative section, declare the following variables:

   1. Variable `today` of type `DATE`. Initialize today with `SYSDATE`.

```
DECLARE
today DATE:=SYSDATE;
```

   2. Variable `tomorrow` of type `today`. Use `%TYPE` attribute to declare this variable.

```
tomorrow today%TYPE;
```

b. In the executable section initialize the variable `tomorrow` with an expression, which calculates tomorrow's date (add one to the value in `today`). Print the value of `today` and `tomorrow` after printing "Hello World."

```
BEGIN
tomorrow:=today +1;
DBMS_OUTPUT.PUT_LINE(' Hello World ');
DBMS_OUTPUT.PUT_LINE('TODAY IS : '|| today);
DBMS_OUTPUT.PUT_LINE('TOMORROW IS : ' || tomorrow);
END;
```

c. Execute and save your script as `lab_02_04_soln.sql`. Follow the instructions in step 2 e) of practice 1 to save the file. Sample output is as follows:

```
Hello World
TODAY IS : 12-JAN-04
TOMORROW IS : 13-JAN-04
PL/SQL procedure successfully completed.
```

5. Edit the `lab_02_04_soln.sql` script.

   a. Add code to create two bind variables.
      Create bind variables `basic_percent` and `pf_percent` of type `NUMBER`.

```
VARIABLE basic_percent NUMBER
VARIABLE pf_percent NUMBER
```

   b. In the executable section of the PL/SQL block assign the values 45 and 12 to `basic_percent` and `pf_percent` respectively.

```
:basic_percent:=45;
:pf_percent:=12;
```

   c. Terminate the PL/SQL block with "/" and display the value of the bind variables by using the `PRINT` command.

```
/
PRINT basic_percent
PRINT pf_percent
```

OR

```
PRINT
```

   d. Execute and Save your script as `lab_02_05_soln.sql`. Sample output is as follows:

```
Hello World
TODAY IS : 12-JAN-04
TOMORROW IS : 13-JAN-04
PL/SQL procedure successfully completed.
```

BASIC_PERCENT
45

Next Page

Click the Next Page button.

PF_PERCENT
12

Practice 3

Note: Use *i*SQL*Plus for this practice.

```
  DECLARE
   weight    NUMBER(3) := 600;
   message   VARCHAR2(255) := 'Product 10012';
  BEGIN
    DECLARE
     weight   NUMBER(3) := 1;
     message   VARCHAR2(255) := 'Product 11001';
1    new_locn  VARCHAR2(50) := 'Europe';
    BEGIN
     weight := weight + 1;
     new_locn := 'Western ' || new_locn;

2   END;
   weight := weight + 1;
   message := message || ' is in stock';
   new_locn := 'Western ' || new_locn;

  END;
```

1. Evaluate the preceding PL/SQL block and determine the data type and value of each of the following variables according to the rules of scoping.

   a. The value of `weight` at position 1 is:
   2
   The data type is `NUMBER`.

   b. The value of `new_locn` at position 1 is:
   Western Europe
   The data type is `VARCHAR2`.

   c. The value of `weight` at position 2 is:
   601
   The data type is `NUMBER`.

   d. The value of `message` at position 2 is:
   Product 10012 is in stock.
   The data type is `VARCHAR2`.

   e. The value of `new_locn` at position 2 is:
   Illegal because `new_locn` is not visible outside the subblock.

```
DECLARE
   customer        VARCHAR2(50) := 'Womansport';
   credit_rating   VARCHAR2(50) := 'EXCELLENT';
BEGIN
    DECLARE
       customer NUMBER(7) := 201;
       name      VARCHAR2(25) := 'Unisports';
    BEGIN
       credit_rating :='GOOD';
       …
    END;
  …
END;
```

2. In the preceding PL/SQL block, determine the values and data types for each of the following cases.

   a. The value of `customer` in the nested block is:
      201
      The data type is `NUMBER`.F

   b. The value of `name` in the nested block is:
      Unisports
      The data type is `VARCHAR2`.

   c. The value of `credit_rating` in the nested block is:
      GOOD
      The data type is `VARCHAR2`.

   d. The value of `customer` in the main block is:
      Womansport
      The data type is `VARCHAR2`.

   e. The value of `name` in the main block is:
      `name` is not visible in the main block and you would see an error.

   f. The value of `credit_rating` in the main block is:
      GOOD
      The data type is `VARCHAR2`.

3. Use the same session that you used to execute the practices in Lesson 2. If you have opened a new session, then execute `lab_02_05_soln.sql`. Edit `lab_02_05_soln.sql`.

   a. Use single line comment syntax to comment the lines that create the bind variables.

```
-- VARIABLE basic_percent NUMBER
-- VARIABLE pf_percent NUMBER
```

   b. Use multiple line comments in the executable section to comment the lines that assign values to the bind variables.

```
/*  :basic_percent:=45;
:pf_percent:=12;  */
```

c. Declare two variables: `fname` of type `VARCHAR2` and size 15, and `emp_sal` of type `NUMBER` and size 10.

```
fname VARCHAR2(15);
emp_sal NUMBER(10);
```

d. Include the following SQL statement in the executable section:

```
SELECT first_name, salary INTO fname, emp_sal
FROM employees WHERE employee_id=110;
```

e. Change the line that prints “Hello World” to print “Hello” and the first name. You can comment the lines that display the dates and print the bind variables, if you want to.

```
DBMS_OUTPUT.PUT_LINE(' Hello '|| fname);
```

f. Calculate the contribution of the employee towards provident fund (PF).
PF is 12% of the basic salary, and the basic salary is 45% of the salary. Use the bind variables for the calculation. Try to use only one expression to calculate the PF. Print the employee’s salary and his contribution toward PF.

```
DBMS_OUTPUT.PUT_LINE('YOUR SALARY IS : '||emp_sal);
DBMS_OUTPUT.PUT_LINE('YOUR CONTRIBUTION TOWARDS PF:
'||emp_sal*:basic_percent/100*:pf_percent/100);
```

g. Execute and save your script as `lab_03_03_soln.sql`. Sample output is as follows:

```
Hello John
YOUR SALARY IS : 8200
YOUR CONTRIBUTION TOWARDS PF: 442.8
PL/SQL procedure successfully completed.
```

4. Accept a value at run time using the substitution variable. In this practice, you will modify the script `lab_03_04.sql` to accept user input.

   a. Load the script `lab_03_04.sql` file.
   b. Include the `PROMPT` command to prompt the user with the following message: “Please enter your employee number.”

```
ACCEPT empno PROMPT 'Please enter your employee number: '
```

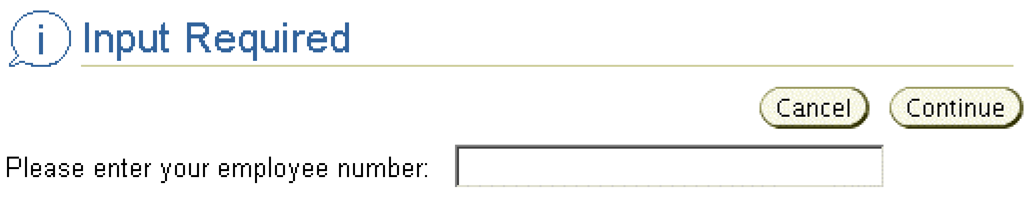
   c. Modify the declaration of the `empno` variable to accept the user input.

```
empno NUMBER(6):=&empno;
```

d. Modify the select statement to include the substitution variable empno.

```
SELECT first_name, salary INTO fname, emp_sal
FROM employees WHERE employee_id=empno;
```

e. Execute and save this script as lab_03_04_soln.sql. Sample output is as follows:



Enter 100 and click the Continue button.

Hello Steven
YOUR SALARY IS : 24000
YOUR CONTRIBUTION TOWARDS PF: 1296
PL/SQL procedure successfully completed.

5. Execute the script lab_03_05.sql. This script creates a table called employee_details.
   a. The employee and employee_details tables have the same data. You will update the data in the employee_details table. Do not update or change the data in the employees table.
   b. Open the script lab_03_05b.sql and observe the code in the file. Note that the code accepts the employee number and the department number from the user.

```
SET SERVEROUTPUT ON
SET VERIFY OFF
ACCEPT emp_id PROMPT 'Please enter your employee number';
ACCEPT emp_deptid PROMPT 'Please enter the department number for which
salary revision is being done';

DECLARE
   emp_authorization NUMBER(5);
   emp_id NUMBER(5):=&emp_id;
   emp_deptid NUMBER(6):=&emp_deptid;
   no_such_employee EXCEPTION;
...
```

   c. You use this as the skeleton script to develop the application, which was discussed in the lesson titled “Introduction.”

Practice 4

Note: Use *i*SQL*Plus for this practice.

1. Create a PL/SQL block that selects the maximum department ID in the `departments` table and stores it in the `max_deptno` variable. Display the maximum department ID.
   a. Declare a variable `max_deptno` of type `NUMBER` in the declarative section.

```
SET SERVEROUTPUT ON
DECLARE
   max_deptno    NUMBER;
```

   b. Start the executable section with the keyword `BEGIN` and include a `SELECT` statement to retrieve the maximum `department_id` from the `departments` table.

```
BEGIN
   SELECT MAX(department_id)  INTO max_deptno  FROM    departments;
```

   c. Display `max_deptno` and end the executable block.

```
DBMS_OUTPUT.PUT_LINE('The maximum department_id is : ' || max_deptno);
END;
```

   d. Execute and save your script as `lab_04_01_soln.sql`. Sample output is as follows:

   The maximum department_id is : 270
   PL/SQL procedure successfully completed.

2. Modify the PL/SQL block you created in exercise 1 to insert a new department into the `departments` table.
   a. Load the script `lab_04_01_soln.sql`. Declare two variables:
   `dept_name` of type `departments.department_name`.
   Bind variable `dept_id` of type `NUMBER`.
   Assign 'Education' to `dept_name` in the declarative section.

```
VARIABLE dept_id NUMBER
…
dept_name departments.department_name%TYPE:= 'Education';
```

   b. You have already retrieved the current maximum department number from the `departments` table. Add 10 to it and assign the result to `dept_id`.

```
:dept_id := 10 + max_deptno;
…
```

c. Include an `INSERT` statement to insert data into the `department_name`, `department_id`, and `location_id` columns of the `departments` table. Use values in `dept_name`, `dept_id` for `department_name`, `department_id` and use NULL for `location_id`.

```
…
INSERT INTO departments (department_id, department_name, location_id)
VALUES (:dept_id,dept_name, NULL);
```

d. Use the SQL attribute `SQL%ROWCOUNT` to display the number of rows that are affected.

```
DBMS_OUTPUT.PUT_LINE (' SQL%ROWCOUNT gives ' || SQL%ROWCOUNT);
…
```

e. Execute a select statement to check if the new department is inserted. You can terminate the PL/SQL block with "/" and include the `SELECT` statement in your script.

```
…
/
SELECT * FROM  departments  WHERE  department_id=:dept_id;
```

f. Execute and save your script as `lab_04_02_soln.sql`. Sample output is as follows:

The maximum department_id is : 270
SQL%ROWCOUNT gives 1
PL/SQL procedure successfully completed.

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
280	Education		

3. In exercise 2, you set `location_id` to null. Create a PL/SQL block that updates the `location_id` to 3000 for the new department. Use the bind variable `dept_id` to update the row.
Note: Skip step a if you have not started a new *i*SQL*Plus session for this practice.

a. If you have started a new *i*SQL*Plus session, delete the department that you have added to the `departments` table and execute the script `lab_04_02_soln.sql`.

```
DELETE FROM departments WHERE department_id=280;
```

b. Start the executable block with the keyword `BEGIN`. Include the `UPDATE` statement to set the `location_id` to 3000 for the new department. Use the bind variable `dept_id` in your `UPDATE` statement.

```
BEGIN
   UPDATE departments SET location_id=3000 WHERE
   department_id=:dept_id;
```

c. End the executable block with the keyword `END`. Terminate the PL/SQL block with "/" and include a `SELECT` statement to display the department that you updated.

```
END;
/
SELECT * FROM departments WHERE department_id=:dept_id;
```

d. Include a `DELETE` statement to delete the department that you added.

```
DELETE FROM departments WHERE department_id=:dept_id;
```

e. Execute and save your script as `lab_04_03_soln.sql`. Sample output is as follows:

PL/SQL procedure successfully completed.

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
280	Education		3000

1 row deleted.

4. Load the script `lab_03_05b.sql` to the *i*SQL*Plus workspace.

a. Observe that the code has nested blocks. You will see the declarative section of the outer block. Look for the comment "INCLUDE EXECUTABLE SECTION OF OUTER BLOCK HERE" and start an executable section.

```
BEGIN
```

b. Include a single `SELECT` statement, which retrieves the `employee_id` of the employee working in the "Human Resources" department. Use the `INTO` clause to store the retrieved value in the variable `emp_authorization`.

```
SELECT employee_id into emp_authorization FROM
   employee_details WHERE department_id=(SELECT department_id
   FROM departments WHERE department_name='Human Resources');
```

c. Save your script as `lab_04_04_soln.sql`.

Practice 5

1. Execute the command in the file `lab_05_01.sql` to create the `messages` table. Write a PL/SQL block to insert numbers into the `messages` table.

    a. Insert the numbers 1 to 10, excluding 6 and 8.
    b. Commit before the end of the block.

```
BEGIN
FOR i in 1..10 LOOP
  IF i = 6 or i = 8 THEN
    null;
  ELSE
    INSERT INTO messages(results)
    VALUES (i);
 END IF;
END LOOP;
COMMIT;
END;
/
```

    c. Execute a `SELECT` statement to verify that your PL/SQL block worked.

```
SELECT * FROM messages;
```

You should see the following output:

RESULTS
1
2
3
4
5
7
9
10

8 rows selected.

2. Execute the script `lab_05_02.sql`. This script creates an `emp` table that is a replica of the `employees` table. It alters the `emp` table to add a new column, `stars`, of `VARCHAR2` data type and size 50. Create a PL/SQL block that inserts an asterisk in the stars column for every $1000 of the employee's salary. Save your script as `lab_05_02_soln.sql`.

    a. Use the `DEFINE` command to define a variable called `empno` and initialize it to 176.

```
SET VERIFY OFF
DEFINE empno = 176
```

b. Start the declarative section of the block and pass the value of empno to the PL/SQL block through an *i*SQL*Plus substitution variable. Declare a variable `asterisk` of type `emp.stars` and initialize it to `NULL`. Create a variable `sal` of type `emp.salary`.

```
DECLARE
  empno        emp.employee_id%TYPE := TO_NUMBER(&empno);
  asterisk     emp.stars%TYPE := NULL;
  sal          emp.salary%TYPE;
```

c. In the executable section, write logic to append an asterisk (*) to the string for every $1000 of the salary. For example, if the employee earns $8000, the string of asterisks should contain eight asterisks. If the employee earns $12500, the string of asterisks should contain 13 asterisks.

```
BEGIN
  SELECT NVL(ROUND(salary/1000), 0)  INTO sal
  FROM emp  WHERE employee_id = empno;

  FOR i IN 1..sal
     LOOP
      asterisk := asterisk ||'*';
      END LOOP;
```

d. Update the `stars` column for the employee with the string of asterisks. Commit before the end of the block.

```
UPDATE emp SET stars = asterisk
      WHERE employee_id = empno;
      COMMIT;
END;
```

e. Display the row from the `emp` table to verify whether your PL/SQL block has executed successfully.

```
SELECT employee_id,salary, stars
FROM emp WHERE employee_id=&empno;
```

f. Execute and save your script as `lab_05_02_soln.sql`. The output is as follows:

EMPLOYEE_ID	SALARY	STARS
176	8600	*********

3. Load the script `lab_04_04_soln.sql`, which you created in exercise 4 of Practice 4.

   a. Look for the comment "INCLUDE SIMPLE IF STATEMENT HERE" and include a simple `IF` statement to check if the values of `emp_id` and `emp_authorization` are the same.

```
IF (emp_id=emp_authorization) THEN
```

   b. Save your script as `lab_05_03_soln.sql`.

Practice 6

1. Write a PL/SQL block to print information about a given country.

    a. Declare a PL/SQL record based on the structure of the `countries` table.

    b. Use the `DEFINE` command to define a variable `countryid`. Assign CA to `countryid`. Pass the value to the PL/SQL block through an *i*SQL*Plus substitution variable.

```
SET SERVEROUTPUT ON
SET VERIFY OFF
DEFINE countryid = CA
```

    c. In the declarative section, use the `%ROWTYPE` attribute and declare the variable `country_record` of type `countries`.

```
DECLARE
 country_record countries%ROWTYPE;
```

    d. In the executable section, get all the information from the `countries` table by using `countryid`. Display selected information about the country. Sample output is as follows:

```
BEGIN
 SELECT    *
 INTO country_record
 FROM countries
 WHERE     country_id = UPPER('&countryid');

 DBMS_OUTPUT.PUT_LINE ('Country Id: ' || country_record.country_id ||
  ' Country Name: ' || country_record.country_name
  || ' Region: ' || country_record.region_id);

END;
```

```
Country Id: CA Country Name: Canada Region: 2
PL/SQL procedure successfully completed.
```

    e. You may want to execute and test the PL/SQL block for the countries with the IDs DE, UK, US.

2. Create a PL/SQL block to retrieve the name of some departments from the `departments` table and print each department name on the screen, incorporating an `INDEX BY` table. Save the script as `lab_06_02_soln.sql`.

   a. Declare an `INDEX BY` table `dept_table_type` of type `departments.department_name`. Declare a variable `my_dept_table` of type `dept_table_type` to temporarily store the name of the departments.

```
SET SERVEROUTPUT ON
DECLARE
   TYPE dept_table_type is table of departments.department_name%TYPE
   INDEX BY PLS_INTEGER;
   my_dept_table   dept_table_type;
```

   b. Declare two variables: `loop_count` and `deptno` of type `NUMBER`. Assign 10 to `loop_count` and 0 to `deptno`.

```
loop_count     NUMBER (2):=10;
deptno         NUMBER (4):=0;
```

   c. Using a loop, retrieve the name of 10 departments and store the names in the `INDEX BY` table. Start with `department_id` 10. Increase `deptno` by 10 for every iteration of the loop. The following table shows the `department_id` for which you should retrieve the `department_name` and store in the `INDEX BY` table.

DEPARTMENT_ID	**DEPARTMENT_NAME**
10	Administration
20	Marketing
30	Purchasing
40	Human Resources
50	Shipping
60	IT
70	Public Relations
80	Sales
90	Executive
100	Finance

```
BEGIN

  FOR i IN 1..loop_count
  LOOP
      deptno:=deptno+10;
     SELECT department_name
     INTO my_dept_table(i)
     FROM departments
     WHERE department_id = deptno;
  END LOOP;
```

d. Using another loop, retrieve the department names from the `INDEX BY` table and display them.

```
FOR i IN 1..loop_count
  LOOP
    DBMS_OUTPUT.PUT_LINE (my_dept_table(i));
  END LOOP;
END;
```

e. Execute and save your script as `lab_06_02_soln.sql`. The output is as follows:

```
Administration
Marketing
Purchasing
Human Resources
Shipping
IT
Public Relations
Sales
Executive
Finance
PL/SQL procedure successfully completed.
```


3. Modify the block that you created in exercise 2 to retrieve all information about each department from the `departments` table and display the information. Use an `INDEX BY` table of records.

   a. Load the script `lab_06_02_soln.sql`.
   b. You have declared the `INDEX BY` table to be of type `departments.department_name`. Modify the declaration of the `INDEX BY` table, to temporarily store the number, name, and location of all the departments. Use the `%ROWTYPE` attribute.

```
SET SERVEROUTPUT ON
DECLARE
   TYPE dept_table_type is table of departments%ROWTYPE
   INDEX BY PLS_INTEGER;
   my_dept_table dept_table_type;
   loop_count          NUMBER (2):=10;
   deptno              NUMBER (4):=0;
```

c. Modify the select statement to retrieve all department information currently in the `departments` table and store it in the `INDEX BY` table.

```
BEGIN
  FOR i IN 1..loop_count
  LOOP
     deptno := deptno + 10;
     SELECT *
     INTO my_dept_table(i)
     FROM departments
     WHERE department_id = deptno;
  END LOOP;
```

d. Using another loop, retrieve the department information from the `INDEX BY` table and display the information. Sample output is as follows:

```
FOR i IN 1..loop_count
  LOOP
    DBMS_OUTPUT.PUT_LINE ('Department Number: ' ||
my_dept_table(i).department_id
     || ' Department Name: ' || my_dept_table(i).department_name
     || ' Manager Id: '||  my_dept_table(i).manager_id
     || ' Location Id: ' || my_dept_table(i).location_id);
  END LOOP;
END;
```

Department Number: 10 Department Name: Administration Manager Id: 200 Location Id: 1700
Department Number: 20 Department Name: Marketing Manager Id: 201 Location Id: 1800
Department Number: 30 Department Name: Purchasing Manager Id: 114 Location Id: 1700
Department Number: 40 Department Name: Human Resources Manager Id: 203 Location Id: 2400
Department Number: 50 Department Name: Shipping Manager Id: 121 Location Id: 1500
Department Number: 60 Department Name: IT Manager Id: 103 Location Id: 1400
Department Number: 70 Department Name: Public Relations Manager Id: 204 Location Id: 2700
Department Number: 80 Department Name: Sales Manager Id: 145 Location Id: 2500
Department Number: 90 Department Name: Executive Manager Id: 100 Location Id: 1700
Department Number: 100 Department Name: Finance Manager Id: 108 Location Id: 1700
PL/SQL procedure successfully completed.

4. Load the script `lab_05_03_soln.sql`.

   a. Look for the comment "DECLARE AN INDEX BY TABLE OF TYPE VARCHAR2(50). CALL IT ename_table_type" and include the declaration.

```
TYPE ename_table_type IS TABLE OF
    VARCHAR2(50) INDEX BY PLS_INTEGER;
```

   b. Look for the comment "DECLARE A VARIABLE ename_table OF TYPE ename_table_type" and include the declaration.

```
ename_table    ename_table_type;
```

   c. Save your script as `lab_06_04_soln.sql`.

Practice 7

1. Create a PL/SQL block that determines the top *n* salaries of the employees.

   a. Execute the script `lab_07_01.sql` to create a new table, `top_salaries`, for storing the salaries of the employees.

   b. Accept a number *n* from the user where *n* represents the number of top *n* earners from the `employees` table. For example, to view the top five salaries, enter 5.
   Note: Use the `DEFINE` command to define a variable `p_num` to provide the value for *n*. Pass the value to the PL/SQL block through an *i*SQL*Plus substitution variable.

```
DELETE FROM top_salaries;
DEFINE p_num = 5
```

   c. In the declarative section, declare two variables: `num` of type `NUMBER` to accept the substitution variable `p_num`, `sal` of type `employees.salary`. Declare a cursor, `emp_cursor` that retrieves the salaries of employees in descending order. Remember that the salaries should not be duplicated.

```
DECLARE
  num          NUMBER(3) := &p_num;
  sal          employees.salary%TYPE;
  CURSOR         emp_cursor IS
    SELECT       distinct salary
    FROM         employees
    ORDER BY     salary DESC;
```

   d. In the executable section, open the loop and fetch top *n* salaries and insert them into `top_salaries` table. You can use a simple loop to operate on the data. Also, try and use `%ROWCOUNT` and `%FOUND` attributes for the exit condition.

```
BEGIN
  OPEN emp_cursor;
  FETCH emp_cursor INTO  sal;
  WHILE emp_cursor%ROWCOUNT <= num AND emp_cursor%FOUND LOOP
          INSERT INTO top_salaries (salary)
          VALUES (sal);
    FETCH emp_cursor INTO sal;
  END LOOP;
  CLOSE emp_cursor;
END;
```

e. After inserting into the top_salaries table, display the rows with a SELECT statement. The output shown represents the five highest salaries in the employees table.

```
/
SELECT * FROM top_salaries;
```

SALARY
24000
17000
14000
13500
13000

f. Test a variety of special cases, such as *n = 0* or where *n* is greater than the number of employees in the employees table. Empty the top_salaries table after each test.

2. Create a PL/SQL block that does the following:

a. Use the DEFINE command to define a variable p_deptno to provide the department ID.

```
SET SERVEROUTPUT ON
SET VERIFY OFF
SET ECHO OFF
DEFINE p_deptno = 10
```

b. In the declarative section, declare a variable deptno of type NUMBER and assign the value of p_deptno.

```
DECLARE
deptno      NUMBER := &p_deptno;
```

c. Declare a cursor, emp_cursor that retrieves the last_name, salary, and manager_id of the employees working in the department specified in deptno.

```
CURSOR  emp_cursor IS
      SELECT     last_name, salary,manager_id
      FROM       employees
      WHERE      department_id = deptno;
```

d. In the executable section use the cursor FOR loop to operate on the data retrieved. If the salary of the employee is less than 5000 and if the manager ID is either 101 or 124, display the message <<*last_name*>> Due for a raise. Otherwise, display the message <<*last_name*>> Not due for a raise.

```
BEGIN
 FOR emp_record IN emp_cursor
  LOOP
   IF emp_record.salary < 5000 AND (emp_record.manager_id=101 OR
emp_record.manager_id=124) THEN
      DBMS_OUTPUT.PUT_LINE (emp_record.last_name || ' Due for a raise');
   ELSE
      DBMS_OUTPUT.PUT_LINE (emp_record.last_name || ' Not Due for a
raise');
   END IF;
  END LOOP;
END;
```

e. Test the PL/SQL block for the following cases:

Department ID	Message
10	Whalen Due for a raise
20	Hartstein Not Due for a raise Fay Not Due for a raise
50	Weiss Not Due for a raise Fripp Not Due for a raise Kaufling Not Due for a raise Vollman Not Due for a raise Mourgas Not Due for a raise . . . . . . Rajs Due for a raise
80	Russel Not Due for a raise Partners Not Due for a raise Errazuriz Not Due for a raise Cambrault Not Due for a raise . . . . . .

3. Write a PL/SQL block, which declares and uses cursors with parameters.
   In a loop, use a cursor to retrieve the department number and the department name from the `departments` table for a department whose `department_id` is less than 100. Pass the department number to another cursor as a parameter to retrieve from the `employees` table the details of employee last name, job, hire date, and salary of those employees whose `employee_id` is less than 120 and who work in that department.

a. In the declarative section declare a cursor `dept_cursor` to retrieve `department_id`, `department_name` for those departments with `department_id` less than 100. Order by `department_id`.

```
SET SERVEROUTPUT ON
DECLARE
     CURSOR dept_cursor IS
       SELECT department_id,department_name
       FROM    departments
       WHERE department_id < 100
       ORDER BY    department_id;
```

b. Declare another cursor `emp_cursor` that takes the department number as parameter and retrieves `last_name`, `job_id`, `hire_date`, and `salary` of those employees with `employee_id` of less than 120 and who work in that department.

```
CURSOR emp_cursor(v_deptno NUMBER) IS
        SELECT last_name,job_id,hire_date,salary
        FROM    employees
        WHERE   department_id = v_deptno
        AND employee_id < 120;
```

c. Declare variables to hold the values retrieved from each cursor. Use the `%TYPE` attribute while declaring variables.

```
current_deptno departments.department_id%TYPE;
current_dname  departments.department_name%TYPE;
ename          employees.last_name%TYPE;
job            employees.job_id%TYPE;
hiredate       employees.hire_date%TYPE;
sal            employees.salary%TYPE;
```

d. Open the `dept_cursor`, use a simple loop and fetch values into the variables declared. Display the department number and department name.

```
BEGIN
OPEN dept_cursor;
      LOOP
           FETCH dept_cursor INTO current_deptno,current_dname;
           EXIT WHEN dept_cursor%NOTFOUND;
           DBMS_OUTPUT.PUT_LINE ('Department Number : ' ||
current_deptno ||  '  Department Name : ' || current_dname);
```

e. For each department, open the `emp_cursor` by passing the current department number as a parameter. Start another loop and fetch the values of `emp_cursor` into variables and print all the details retrieved from the `employees` table.
Note: You may want to print a line after you have displayed the details of each department. Use appropriate attributes for the exit condition. Also check if a cursor is already open before opening the cursor.

```
IF emp_cursor%ISOPEN THEN
      CLOSE emp_cursor;
END IF;
  OPEN emp_cursor (current_deptno);
  LOOP
    FETCH emp_cursor INTO  ename,job,hiredate,sal;
    EXIT WHEN emp_cursor%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE (ename || '    ' ||  job || '   '  ||  hiredate
|| '   ' ||  sal);
  END LOOP;
 DBMS_OUTPUT.PUT_LINE('-------------------------------------------------------
--------------------------------------');
  CLOSE emp_cursor;
```

f. Close all the loops and cursors, and end the executable section. Execute the script.

```
END LOOP;
     CLOSE dept_cursor;
END;
```

The sample output is as follows:

```
Department Number : 10 Department Name : Administration
--------------------------------------------------------------------------------
Department Number : 20 Department Name : Marketing
--------------------------------------------------------------------------------
Department Number : 30 Department Name : Purchasing
Raphaely PU_MAN 07-DEC-94 11000
Khoo PU_CLERK 18-MAY-95 3100
Baida PU_CLERK 24-DEC-97 2900
Tobias PU_CLERK 24-JUL-97 2800
Himuro PU_CLERK 15-NOV-98 2600
Colmenares PU_CLERK 10-AUG-99 2500
--------------------------------------------------------------------------------
Department Number : 40 Department Name : Human Resources
--------------------------------------------------------------------------------
Department Number : 50 Department Name : Shipping
--------------------------------------------------------------------------------
Department Number : 60 Department Name : IT
Hunold IT_PROG 03-JAN-90 9000
Ernst IT_PROG 21-MAY-91 6000
Austin IT_PROG 25-JUN-97 4800
Pataballa IT_PROG 05-FEB-98 4800
Lorentz IT_PROG 07-FEB-99 4200
--------------------------------------------------------------------------------
Department Number : 70 Department Name : Public Relations
--------------------------------------------------------------------------------
Department Number : 80 Department Name : Sales
--------------------------------------------------------------------------------
Department Number : 90 Department Name : Executive
King AD_PRES 17-JUN-87 24000
Kochhar AD_VP 21-SEP-89 17000
De Haan AD_VP 13-JAN-93 17000
--------------------------------------------------------------------------------
PL/SQL procedure successfully completed.
```

4. Load the script `lab_06_04_soln.sql`.

   a. Look for the comment "DECLARE A CURSOR CALLED emp_records TO HOLD salary, first_name, and last_name of employees" and include the declaration. Create the cursor such that it retrieves the `salary`, `first_name`, and `last_name` of employees in the department specified by the user (substitution variable `emp_deptid`). Use the `FOR UPDATE` clause.

```
CURSOR emp_records IS SELECT salary,first_name,last_name
FROM employee_details WHERE department_id=emp_deptid
FOR UPDATE;
```

   b. Look for the comment "INCLUDE EXECUTABLE SECTION OF INNER BLOCK HERE" and start the executable block.

```
BEGIN
```

   c. Only employees working in the departments with `department_id` 20, 60, 80,100, and 110 are eligible for raises this quarter. Check if the user has entered any of these department IDs. If the value does not match, display the message "SORRY, NO SALARY REVISIONS FOR EMPLOYEES IN THIS DEPARTMENT." If the value matches, open the cursor `emp_records`.

```
IF (emp_deptid NOT IN (20,60,80,100,110)) THEN
DBMS_OUTPUT.PUT_LINE ('SORRY, NO SALARY REVISIONS FOR
EMPLOYEES IN THIS DEPARTMENT');
 ELSE
   OPEN emp_records;
```

   d. Start a simple loop and fetch the values into `emp_sal`, `emp_fname`, and `emp_lname`. Use `%NOTFOUND` for the exit condition.

```
 LOOP
   FETCH emp_records INTO emp_sal,emp_fname,emp_lname;
   EXIT WHEN emp_records%NOTFOUND;
```

   e. Include a `CASE` expression. Use the following table as reference for the conditions in the `WHEN` clause of the `CASE` expression.
   Note: In your `CASE` expressions use the constants such as `c_range1`, `c_hike1` that are already declared.

salary	**Hike percentage**
< 6500	20
> 6500 < 9500	15
> 9500 <12000	8
>12000	3

For example, if the salary of the employee is less than 6500, then increase the salary by 20 percent. In every `WHEN` clause, concatenate the `first_name` and `last_name` of the employee and store it in the `INDEX BY` table. Increment the value in variable `i` so that you can store the string in the next location. Include an `UPDATE` statement with the `WHERE CURRENT OF` clause.

```
CASE
    WHEN   emp_sal<c_range1 THEN
    ename_table(i):=emp_fname||' '||emp_lname;
    i:=i+1;
    UPDATE employee_details SET salary=emp_sal +  (emp_sal*c_hike1)
    WHERE CURRENT OF emp_records;
    WHEN   emp_sal<c_range2 THEN
     ename_table(i):=emp_fname||' '||emp_lname;
     i:=i+1;
     UPDATE employee_details SET salary=emp_sal+(emp_sal*c_hike2)
     WHERE CURRENT OF emp_records;
    WHEN (emp_sal<c_range3) THEN
     ename_table(i):=emp_fname||' '||emp_lname;
     i:=i+1;
     UPDATE employee_details SET salary=emp_sal+(emp_sal*c_hike3)
     WHERE CURRENT OF emp_records;
    ELSE
    ename_table(i):=emp_fname||' '||emp_lname;
    i:=i+1;
     UPDATE employee_details SET salary=emp_sal+(emp_sal*c_hike4)
     WHERE CURRENT OF emp_records;
    END CASE;
```

f. Close the loop. Use the `%ROWCOUNT` attribute and print the number of records that were modified. Close the cursor.

```
END LOOP;

DBMS_OUTPUT.PUT_LINE ('NUMBER OF RECORDS MODIFIED :
'||emp_records%ROWCOUNT);
CLOSE emp_records;
```

g. Include a simple loop to print the names of all the employees whose salaries were revised.
Note: You already have the names of these employees in the `INDEX BY` table.
Look for the comment “CLOSE THE INNER BLOCK” and include an `END IF` statement and an `END` statement.

```
DBMS_OUTPUT.PUT_LINE ('The following employees'' salaries are updated');
 FOR i IN ename_table.FIRST..ename_table.LAST
  LOOP
     DBMS_OUTPUT.PUT_LINE(ename_table(i));
  END LOOP;
END IF;
END;
```

h. Save your script as `lab_07_04_soln.sql`.

Practice 8

1. The purpose of this example is to show the usage of predefined exceptions. Write a PL/SQL block to select the name of the employee with a given salary value.

   a. Delete all the records in the `messages` table. Use the `DEFINE` command to define a variable `sal` and initialize it to 6000.

```
DELETE FROM MESSAGES;
SET VERIFY OFF
DEFINE sal = 6000
```

   b. In the declarative section declare two variables: `ename` of type `employees.last_name` and `emp_sal` of type `employees.salary`. Pass the value of the substitution variables to `emp_sal`.

```
DECLARE
  ename    employees.last_name%TYPE;
  emp_sal  employees.salary%TYPE := &sal;
```

   c. In the executable section retrieve the last names of employees whose salaries are equal to the value in `emp_sal`.
   Note: Do not use explicit cursors.
   If the salary entered returns only one row, insert into the `messages` table the employee's name and the salary amount.

```
BEGIN
  SELECT   last_name
  INTO     ename
  FROM     employees
  WHERE    salary = emp_sal;
  INSERT INTO messages (results)
  VALUES (ename || ' - ' || emp_sal);
```

   d. If the salary entered does not return any rows, handle the exception with an appropriate exception handler and insert into the `messages` table the message "No employee with a salary of <*salary*>."

```
EXCEPTION
  WHEN no_data_found THEN
    INSERT INTO messages (results)
    VALUES ('No employee with a salary of '|| TO_CHAR(emp_sal));
```

   e. If the salary entered returns more than one row, handle the exception with an appropriate exception handler and insert into the `messages` table the message "More than one employee with a salary of <*salary*>."

```
WHEN too_many_rows THEN
    INSERT INTO messages (results)
```

```
    VALUES ('More than one employee with a salary of '||
            TO_CHAR(emp_sal));
```

f. Handle any other exception with an appropriate exception handler and insert into the `messages` table the message "Some other error occurred."

```
WHEN others THEN
    INSERT INTO messages (results)
    VALUES ('Some other error occurred.');
END;
```

g. Display the rows from the `messages` table to check whether the PL/SQL block has executed successfully. Sample output is as follows:

```
/
SELECT * FROM messages;
```

RESULTS
More than one employee with a salary of 6000

2. The purpose of this example is to show how to declare exceptions with a standard Oracle Server error. Use the Oracle server error ORA-02292 (integrity constraint violated – child record found).

a. In the declarative section declare an exception `childrecord_exists`. Associate the declared exception with the standard Oracle server error –02292.

```
SET SERVEROUTPUT ON
DECLARE
   childrecord_exists EXCEPTION;
   PRAGMA EXCEPTION_INIT(childrecord_exists, -02292);
```

b. In the executable section display "Deleting department 40.....". Include a `DELETE` statement to delete the department with `department_id` 40.

```
BEGIN
   DBMS_OUTPUT.PUT_LINE(' Deleting department 40........');
   delete from departments where department_id=40;
```

c. Include an exception section to handle the `childrecord_exists` exception and display the appropriate message. Sample output is as follows:

```
EXCEPTION
   WHEN childrecord_exists THEN
   DBMS_OUTPUT.PUT_LINE(' Cannot delete this department. There are
employees in this department (child records exist.)');
END;
```

```
Deleting department 40........
Cannot delete this department. There are employees in this department
(child records exist.)
PL/SQL procedure successfully completed.
```

3. Load the script `lab_07_04_soln.sql`.
   a. Observe the declarative section of the outer block. Note that the `no_such_employee` exception is declared.
   b. Look for the comment "RAISE EXCEPTION HERE." If the value of `emp_id` is not between 100 and 206, then raise the `no_such_employee` exception.

```
  IF (emp_id NOT BETWEEN 100 AND 206) THEN
   RAISE no_such_employee;
  END IF;
```

   c. Look for the comment "INCLUDE EXCEPTION SECTION FOR OUTER BLOCK" and handle the exceptions `no_such_employee` and `too_many_rows`. Display appropriate messages when the exceptions occur. The `employees` table has only one employee working in the `HR` department and therefore the code is written accordingly. The `too_many_rows` exception is handled to indicate that the select statement retrieves more than one employee working in the `HR` department.

```
EXCEPTION
  WHEN no_such_employee THEN
  DBMS_OUTPUT.PUT_LINE ('NO EMPLOYEE EXISTS WITH THE
  GIVEN EMPLOYEE NUMBER: PLEASE CHECK');

  WHEN TOO_MANY_ROWS THEN
  DBMS_OUTPUT.PUT_LINE  (' THERE IS MORE THAN ONE
  EMPLOYEE IN THE HR DEPARTMENT. ');
```

   d. Close the outer block.

```
END;
```

   e. Save your script as `lab_08_03_soln.sql`.
   f. Execute the script. Enter the employee number and the department number and observe the output. Enter different values and check for different conditions. The sample output for employee ID 203 and department ID 100 is as follows:

```
NUMBER OF RECORDS MODIFIED : 6
The following employees' salaries are updated
Nancy Greenberg
Daniel Faviet
John Chen
Ismael Sciarra
Jose Manuel Urman
Luis Popp
PL/SQL procedure successfully completed.
```

Practice 9

1. In *i*SQL*Plus, load the script `lab_02_04_soln.sql` that you created for exercise 4 of practice 2.
   a. Modify the script to convert the anonymous block to a procedure called `greet`.

```
CREATE PROCEDURE greet IS
   today DATE:=SYSDATE;
   tomorrow today%TYPE;
...
```

   b. Execute the script to create the procedure.
   c. Save this script as `lab_09_01_soln.sql`.
   d. Click the Clear button to clear the workspace.
   e. Create and execute an anonymous block to invoke the procedure `greet`. Sample output is as follows:

```
BEGIN
 greet;
END;
```

```
Hello World
TODAY IS : 20-JAN-04
TOMORROW IS : 21-JAN-04
PL/SQL procedure successfully completed.
```

2. Load the script `lab_09_01_soln.sql`.
   a. Drop the procedure `greet` by issuing the following command:

```
DROP PROCEDURE greet
```

   b. Modify the procedure to accept an argument of type `VARCHAR2`. Call the argument `name`.

```
CREATE PROCEDURE greet(name VARCHAR2) IS
   today DATE:=SYSDATE;
   tomorrow today%TYPE;
```

   c. Print `Hello` *`<name>`* instead of printing `Hello World`.

```
BEGIN
   tomorrow:=today +1;
   DBMS_OUTPUT.PUT_LINE(' Hello '|| name);
```

   d. Save your script as `lab_09_02_soln.sql`.
   e. Execute the script to create the procedure.

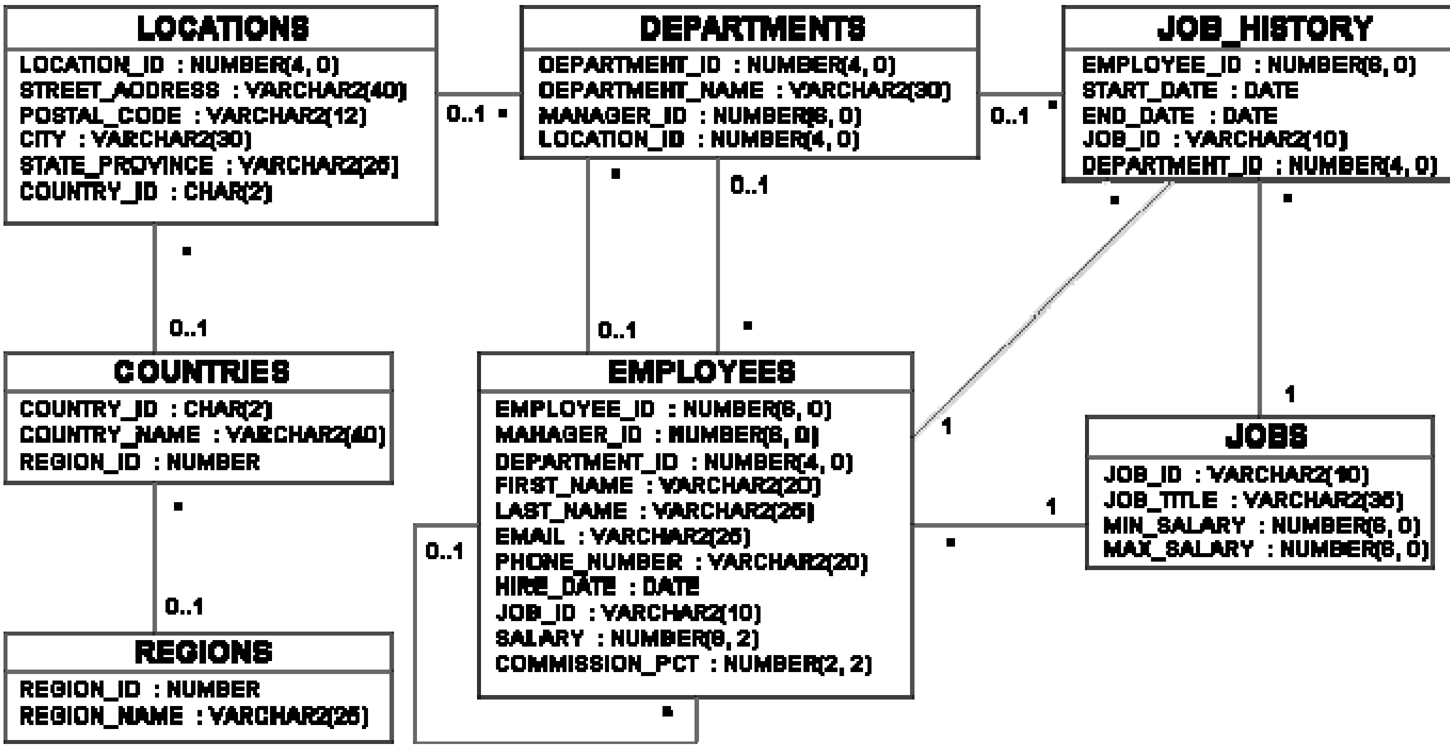
f. Create and execute an anonymous block to invoke the procedure `greet` with a parameter. Sample output is as follows:

```
BEGIN
 greet('Neema');
END;
```

```
Hello Neema
TODAY IS : 20-JAN-04
TOMORROW IS : 21-JAN-04
PL/SQL procedure successfully completed.
```

B
Table Descriptions and Data

ENTITY RELATIONSHIP DIAGRAM



Tables in the Schema

```
SELECT * FROM tab;
```

TNAME	TABTYPE	CLUSTERID
COUNTRIES	TABLE	
DEPARTMENTS	TABLE	
EMPLOYEES	TABLE	
EMP_DETAILS_VIEW	VIEW	
JOBS	TABLE	
JOB_HISTORY	TABLE	
LOCATIONS	TABLE	
REGIONS	TABLE	

8 rows selected.

`regions` Table

```
DESCRIBE regions
```

Name	Null?	Type
REGION_ID	NOT NULL	NUMBER
REGION_NAME		VARCHAR2(25)

```
SELECT * FROM regions;
```

REGION_ID	REGION_NAME
1	Europe
2	Americas
3	Asia
4	Middle East and Africa

countries Table

```
DESCRIBE countries
```

Name	Null?	Type
COUNTRY_ID	NOT NULL	CHAR(2)
COUNTRY_NAME		VARCHAR2(40)
REGION_ID		NUMBER

```
SELECT * FROM countries;
```

CO	COUNTRY_NAME	REGION_ID
AR	Argentina	2
AU	Australia	3
BE	Belgium	1
BR	Brazil	2
CA	Canada	2
CH	Switzerland	1
CN	China	3
DE	Germany	1
DK	Denmark	1
EG	Egypt	4
FR	France	1
HK	HongKong	3
IL	Israel	4
IN	India	3
CO	COUNTRY_NAME	REGION_ID
IT	Italy	1
JP	Japan	3
KW	Kuwait	4
MX	Mexico	2
NG	Nigeria	4
NL	Netherlands	1
SG	Singapore	3
UK	United Kingdom	1
US	United States of America	2
ZM	Zambia	4
ZW	Zimbabwe	4

25 rows selected.

`locations` Table

```
DESCRIBE locations;
```

Name	Null?	Type
LOCATION_ID	NOT NULL	NUMBER(4)
STREET_ADDRESS		VARCHAR2(40)
POSTAL_CODE		VARCHAR2(12)
CITY	NOT NULL	VARCHAR2(30)
STATE_PROVINCE		VARCHAR2(25)
COUNTRY_ID		CHAR(2)

```
SELECT * FROM locations;
```

LOCATION_ID	STREET_ADDRESS	POSTAL_CODE	CITY	STATE_PROVINCE	CO
1000	1297 Via Cola di Rie	00989	Roma		IT
1100	93091 Calle della Testa	10934	Venice		IT
1200	2017 Shinjuku-ku	1689	Tokyo	Tokyo Prefecture	JP
1300	9450 Kamiya-cho	6823	Hiroshima		JP
1400	2014 Jabberwocky Rd	26192	Southlake	Texas	US
1500	2011 Interiors Blvd	99236	South San Francisco	California	US
1600	2007 Zagora St	50090	South Brunswick	New Jersey	US
1700	2004 Charade Rd	98199	Seattle	Washington	US
1800	147 Spadina Ave	M5V 2L7	Toronto	Ontario	CA
1900	6092 Boxwood St	YSW 9T2	Whitehorse	Yukon	CA
2000	40-5-12 Laogianggen	190518	Beijing		CN
2100	1298 Vileparle (E)	490231	Bombay	Maharashtra	IN
LOCATION_ID	STREET_ADDRESS	POSTAL_CODE	CITY	STATE_PROVINCE	CO
2400	8204 Arthur St		London		UK
2500	Magdalen Centre, The Oxford Science Park	OX9 9ZB	Oxford	Oxford	UK
2600	9702 Chester Road	09629850293	Stretford	Manchester	UK
2700	Schwanthalerstr. 7031	80925	Munich	Bavaria	DE
2800	Rua Frei Caneca 1360	01307-002	Sao Paulo	Sao Paulo	BR
2900	20 Rue des Corps-Saints	1730	Geneva	Geneve	CH
3000	Murtenstrasse 921	3095	Bern	BE	CH
3100	Pieter Breughelstraat 837	3029SK	Utrecht	Utrecht	NL
3200	Mariano Escobedo 9991	11932	Mexico City	Distrito Federal,	MX

23 rows selected.

`departments` Table

```
DESCRIBE departments
```

Name	Null?	Type
DEPARTMENT_ID	NOT NULL	NUMBER(4)
DEPARTMENT_NAME	NOT NULL	VARCHAR2(30)
MANAGER_ID		NUMBER(6)
LOCATION_ID		NUMBER(4)

```
SELECT * FROM departments;
```

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
10	Administration	200	1700
20	Marketing	201	1800
30	Purchasing	114	1700
40	Human Resources	203	2400
50	Shipping	121	1500
60	IT	103	1400
70	Public Relations	204	2700
80	Sales	145	2500
90	Executive	100	1700
100	Finance	108	1700
110	Accounting	205	1700
120	Treasury		1700
130	Corporate Tax		1700
140	Control And Credit		1700
DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
150	Shareholder Services		1700
160	Benefits		1700
170	Manufacturing		1700
180	Construction		1700
190	Contracting		1700
200	Operations		1700
210	IT Support		1700
220	NOC		1700
230	IT Helpdesk		1700
240	Government Sales		1700
250	Retail Sales		1700
260	Recruiting		1700
270	Payroll		1700

27 rows selected.

jobs Table

```
DESCRIBE jobs
```

Name	Null?	Type
JOB_ID	NOT NULL	VARCHAR2(10)
JOB_TITLE	NOT NULL	VARCHAR2(35)
MIN_SALARY		NUMBER(6)
MAX_SALARY		NUMBER(6)

```
SELECT * FROM jobs;
```

JOB_ID	JOB_TITLE	MIN_SALARY	MAX_SALARY
AD_PRES	President	20000	40000
AD_VP	Administration Vice President	15000	30000
AD_ASST	Administration Assistant	3000	6000
FI_MGR	Finance Manager	8200	16000
FI_ACCOUNT	Accountant	4200	9000
AC_MGR	Accounting Manager	8200	16000
AC_ACCOUNT	Public Accountant	4200	9000
SA_MAN	Sales Manager	10000	20000
SA_REP	Sales Representative	6000	12000
PU_MAN	Purchasing Manager	8000	15000
PU_CLERK	Purchasing Clerk	2500	5500
ST_MAN	Stock Manager	5500	8500
ST_CLERK	Stock Clerk	2000	5000
SH_CLERK	Shipping Clerk	2500	5500
JOB_ID	JOB_TITLE	MIN_SALARY	MAX_SALARY
IT_PROG	Programmer	4000	10000
MK_MAN	Marketing Manager	9000	15000
MK_REP	Marketing Representative	4000	9000
HR_REP	Human Resources Representative	4000	9000
PR_REP	Public Relations Representative	4500	10500

19 rows selected.

`employees` Table

```
DESCRIBE employees
```

Name	Null?	Type
EMPLOYEE_ID	NOT NULL	NUMBER(6)
FIRST_NAME		VARCHAR2(20)
LAST_NAME	NOT NULL	VARCHAR2(25)
EMAIL	NOT NULL	VARCHAR2(25)
PHONE_NUMBER		VARCHAR2(20)
HIRE_DATE	NOT NULL	DATE
JOB_ID	NOT NULL	VARCHAR2(10)
SALARY		NUMBER(8,2)
COMMISSION_PCT		NUMBER(2,2)
MANAGER_ID		NUMBER(6)
DEPARTMENT_ID		NUMBER(4)

`employees` Table (continued)

The headings for the `commission_pct`, `manager_id`, and `department_id` columns are set to `comm`, `mgrid`, and `deptid` in the following screenshot to fit the table values across the page.

```
SELECT * FROM employees;
```

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	comm	mgrid	deptid
100	Steven	King	SKING	515.123.4567	17-JUN-87	AD_PRES	24000			90
101	Neena	Kochhar	NKOCHHAR	515.123.4568	21-SEP-89	AD_VP	17000		100	90
102	Lex	De Haan	LDEHAAN	515.123.4569	13-JAN-93	AD_VP	17000		100	90
103	Alexander	Hunold	AHUNOLD	590.423.4567	03-JAN-90	IT_PROG	9000		102	60
104	Bruce	Ernst	BERNST	590.423.4568	21-MAY-91	IT_PROG	6000		103	60
105	David	Austin	DAUSTIN	590.423.4569	25-JUN-97	IT_PROG	4800		103	60
106	Valli	Pataballa	VPATABAL	590.423.4560	05-FEB-98	IT_PROG	4800		103	60
107	Diana	Lorentz	DLORENTZ	590.423.5567	07-FEB-99	IT_PROG	4200		103	60
108	Nancy	Greenberg	NGREENBE	515.124.4569	17-AUG-94	FI_MGR	12000		101	100
109	Daniel	Faviet	DFAVIET	515.124.4169	16-AUG-94	FI_ACCOUNT	9000		108	100
110	John	Chen	JCHEN	515.124.4269	28-SEP-97	FI_ACCOUNT	8200		108	100
111	Ismael	Sciarra	ISCIARRA	515.124.4369	30-SEP-97	FI_ACCOUNT	7700		108	100
112	Jose Manuel	Urman	JMURMAN	515.124.4469	07-MAR-98	FI_ACCOUNT	7800		108	100
113	Luis	Popp	LPOPP	515.124.4567	07-DEC-99	FI_ACCOUNT	6900		108	100
EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	comm	mgrid	deptid
114	Den	Raphaely	DRAPHEAL	515.127.4561	07-DEC-94	PU_MAN	11000		100	30
115	Alexander	Khoo	AKHOO	515.127.4562	18-MAY-95	PU_CLERK	3100		114	30
116	Shelli	Baida	SBAIDA	515.127.4563	24-DEC-97	PU_CLERK	2900		114	30
117	Sigal	Tobias	STOBIAS	515.127.4564	24-JUL-97	PU_CLERK	2800		114	30
118	Guy	Himuro	GHIMURO	515.127.4565	15-NOV-98	PU_CLERK	2600		114	30
119	Karen	Colmenares	KCOLMENA	515.127.4566	10-AUG-99	PU_CLERK	2500		114	30
120	Matthew	Weiss	MWEISS	650.123.1234	18-JUL-96	ST_MAN	8000		100	50
121	Adam	Fripp	AFRIPP	650.123.2234	10-APR-97	ST_MAN	8200		100	50
122	Payam	Kaufling	PKAUFLIN	650.123.3234	01-MAY-95	ST_MAN	7900		100	50
123	Shanta	Vollman	SVOLLMAN	650.123.4234	10-OCT-97	ST_MAN	6500		100	50
124	Kevin	Mourgos	KMOURGOS	650.123.5234	16-NOV-99	ST_MAN	5800		100	50
125	Julia	Nayer	JNAYER	650.124.1214	16-JUL-97	ST_CLERK	3200		120	50
126	Irene	Mikkilineni	IMIKKILI	650.124.1224	28-SEP-98	ST_CLERK	2700		120	50
127	James	Landry	JLANDRY	650.124.1334	14-JAN-99	ST_CLERK	2400		120	50

`employees` Table (continued)

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	comm	mgrid	deptid
128	Steven	Markle	SMARKLE	650.124.1434	08-MAR-00	ST_CLERK	2200		120	50
129	Laura	Bissot	LBISSOT	650.124.5234	20-AUG-97	ST_CLERK	3300		121	50
130	Mozhe	Atkinson	MATKINSO	650.124.6234	30-OCT-97	ST_CLERK	2800		121	50
131	James	Marlow	JAMRLOW	650.124.7234	16-FEB-97	ST_CLERK	2500		121	50
132	TJ	Olson	TJOLSON	650.124.8234	10-APR-99	ST_CLERK	2100		121	50
133	Jason	Mallin	JMALLIN	650.127.1934	14-JUN-96	ST_CLERK	3300		122	50
134	Michael	Rogers	MROGERS	650.127.1834	26-AUG-98	ST_CLERK	2900		122	50
135	Ki	Gee	KGEE	650.127.1734	12-DEC-99	ST_CLERK	2400		122	50
136	Hazel	Philtanker	HPHILTAN	650.127.1634	06-FEB-00	ST_CLERK	2200		122	50
137	Renske	Ladwig	RLADWIG	650.121.1234	14-JUL-95	ST_CLERK	3600		123	50
138	Stephen	Stiles	SSTILES	650.121.2034	26-OCT-97	ST_CLERK	3200		123	50
139	John	Seo	JSEO	650.121.2019	12-FEB-98	ST_CLERK	2700		123	50
140	Joshua	Patel	JPATEL	650.121.1834	06-APR-98	ST_CLERK	2500		123	50
141	Trenna	Rajs	TRAJS	650.121.8009	17-OCT-95	ST_CLERK	3500		124	50
EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	comm	mgrid	deptid
142	Curtis	Davies	CDAVIES	650.121.2994	29-JAN-97	ST_CLERK	3100		124	50
143	Randall	Matos	RMATOS	650.121.2874	15-MAR-98	ST_CLERK	2600		124	50
144	Peter	Vargas	PVARGAS	650.121.2004	09-JUL-98	ST_CLERK	2500		124	50
145	John	Russell	JRUSSEL	011.44.1344.429268	01-OCT-96	SA_MAN	14000	.4	100	80
146	Karen	Partners	KPARTNER	011.44.1344.467268	05-JAN-97	SA_MAN	13500	.3	100	80
147	Alberto	Errazuriz	AERRAZUR	011.44.1344.429278	10-MAR-97	SA_MAN	12000	.3	100	80
148	Gerald	Cambrault	GCAMBRAU	011.44.1344.619268	15-OCT-99	SA_MAN	11000	.3	100	80
149	Eleni	Zlotkey	EZLOTKEY	011.44.1344.429018	29-JAN-00	SA_MAN	10500	.2	100	80
150	Peter	Tucker	PTUCKER	011.44.1344.129268	30-JAN-97	SA_REP	10000	.3	145	80
151	David	Bernstein	DBERNSTE	011.44.1344.345268	24-MAR-97	SA_REP	9500	.25	145	80
152	Peter	Hall	PHALL	011.44.1344.478968	20-AUG-97	SA_REP	9000	.25	145	80
153	Christopher	Olsen	COLSEN	011.44.1344.498718	30-MAR-98	SA_REP	8000	.2	145	80
154	Nanette	Cambrault	NCAMBRAU	011.44.1344.987668	09-DEC-98	SA_REP	7500	.2	145	80
155	Oliver	Tuvault	OTUVAULT	011.44.1344.486508	23-NOV-99	SA_REP	7000	.15	145	80
EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	comm	mgrid	deptid
156	Janette	King	JKING	011.44.1345.429268	30-JAN-96	SA_REP	10000	.35	146	80
157	Patrick	Sully	PSULLY	011.44.1345.929268	04-MAR-96	SA_REP	9500	.35	146	80
158	Allan	McEwen	AMCEWEN	011.44.1345.829268	01-AUG-96	SA_REP	9000	.35	146	80
159	Lindsey	Smith	LSMITH	011.44.1345.729268	10-MAR-97	SA_REP	8000	.3	146	80
160	Louise	Doran	LDORAN	011.44.1345.629268	15-DEC-97	SA_REP	7500	.3	146	80
161	Sarath	Sewall	SSEWALL	011.44.1345.529268	03-NOV-98	SA_REP	7000	.25	146	80
162	Clara	Vishney	CVISHNEY	011.44.1346.129268	11-NOV-97	SA_REP	10500	.25	147	80
163	Danielle	Greene	DGREENE	011.44.1346.229268	19-MAR-99	SA_REP	9500	.15	147	80
164	Mattea	Marvins	MMARVINS	011.44.1346.329268	24-JAN-00	SA_REP	7200	.1	147	80
165	David	Lee	DLEE	011.44.1346.529268	23-FEB-00	SA_REP	6800	.1	147	80
166	Sundar	Ande	SANDE	011.44.1346.629268	24-MAR-00	SA_REP	6400	.1	147	80
167	Amit	Banda	ABANDA	011.44.1346.729268	21-APR-00	SA_REP	6200	.1	147	80
168	Lisa	Ozer	LOZER	011.44.1343.929268	11-MAR-97	SA_REP	11500	.25	148	80
169	Harrison	Bloom	HBLOOM	011.44.1343.829268	23-MAR-98	SA_REP	10000	.2	148	80

`employees` Table (continued)

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	comm	mgrid	deptid
170	Tayler	Fox	TFOX	011.44.1343.729268	24-JAN-98	SA_REP	9600	.2	148	80
171	William	Smith	WSMITH	011.44.1343.629268	23-FEB-99	SA_REP	7400	.15	148	80
172	Elizabeth	Bates	EBATES	011.44.1343.529268	24-MAR-99	SA_REP	7300	.15	148	80
173	Sundita	Kumar	SKUMAR	011.44.1343.329268	21-APR-00	SA_REP	6100	.1	148	80
174	Ellen	Abel	EABEL	011.44.1644.429267	11-MAY-96	SA_REP	11000	.3	149	80
175	Alyssa	Hutton	AHUTTON	011.44.1644.429266	19-MAR-97	SA_REP	8800	.25	149	80
176	Jonathon	Taylor	JTAYLOR	011.44.1644.429265	24-MAR-98	SA_REP	8600	.2	149	80
177	Jack	Livingston	JLIVINGS	011.44.1644.429264	23-APR-98	SA_REP	8400	.2	149	80
178	Kimberely	Grant	KGRANT	011.44.1644.429263	24-MAY-99	SA_REP	7000	.15	149	
179	Charles	Johnson	CJOHNSON	011.44.1644.429262	04-JAN-00	SA_REP	6200	.1	149	80
180	Winston	Taylor	WTAYLOR	650.507.9876	24-JAN-98	SH_CLERK	3200		120	50
181	Jean	Fleaur	JFLEAUR	650.507.9877	23-FEB-98	SH_CLERK	3100		120	50
182	Martha	Sullivan	MSULLIVA	650.507.9878	21-JUN-99	SH_CLERK	2500		120	50
183	Girard	Geoni	GGEONI	650.507.9879	03-FEB-00	SH_CLERK	2800		120	50
EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	comm	mgrid	deptid
184	Nandita	Sarchand	NSARCHAN	650.509.1876	27-JAN-96	SH_CLERK	4200		121	50
185	Alexis	Bull	ABULL	650.509.2876	20-FEB-97	SH_CLERK	4100		121	50
186	Julia	Dellinger	JDELLING	650.509.3876	24-JUN-98	SH_CLERK	3400		121	50
187	Anthony	Cabrio	ACABRIO	650.509.4876	07-FEB-99	SH_CLERK	3000		121	50
188	Kelly	Chung	KCHUNG	650.505.1876	14-JUN-97	SH_CLERK	3800		122	50
189	Jennifer	Dilly	JDILLY	650.505.2876	13-AUG-97	SH_CLERK	3600		122	50
190	Timothy	Gates	TGATES	650.505.3876	11-JUL-98	SH_CLERK	2900		122	50
191	Randall	Perkins	RPERKINS	650.505.4876	19-DEC-99	SH_CLERK	2500		122	50
192	Sarah	Bell	SBELL	650.501.1876	04-FEB-96	SH_CLERK	4000		123	50
193	Britney	Everett	BEVERETT	650.501.2876	03-MAR-97	SH_CLERK	3900		123	50
194	Samuel	McCain	SMCCAIN	650.501.3876	01-JUL-98	SH_CLERK	3200		123	50
195	Vance	Jones	VJONES	650.501.4876	17-MAR-99	SH_CLERK	2800		123	50
196	Alana	Walsh	AWALSH	650.507.9811	24-APR-98	SH_CLERK	3100		124	50
197	Kevin	Feeney	KFEENEY	650.507.9822	23-MAY-98	SH_CLERK	3000		124	50
EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	comm	mgrid	deptid
198	Donald	OConnell	DOCONNEL	650.507.9833	21-JUN-99	SH_CLERK	2600		124	50
199	Douglas	Grant	DGRANT	650.507.9844	13-JAN-00	SH_CLERK	2600		124	50
200	Jennifer	Whalen	JWHALEN	515.123.4444	17-SEP-87	AD_ASST	4400		101	10
201	Michael	Hartstein	MHARTSTE	515.123.5555	17-FEB-96	MK_MAN	13000		100	20
202	Pat	Fay	PFAY	603.123.6666	17-AUG-97	MK_REP	6000		201	20
203	Susan	Mavris	SMAVRIS	515.123.7777	07-JUN-94	HR_REP	6500		101	40
204	Hermann	Baer	HBAER	515.123.8888	07-JUN-94	PR_REP	10000		101	70
205	Shelley	Higgins	SHIGGINS	515.123.8080	07-JUN-94	AC_MGR	12000		101	110
206	William	Gietz	WGIETZ	515.123.8181	07-JUN-94	AC_ACCOUNT	8300		205	110

107 rows selected.

`job_history` Table

```
DESCRIBE job_history
```

Name	Null?	Type
EMPLOYEE_ID	NOT NULL	NUMBER(6)
START_DATE	NOT NULL	DATE
END_DATE	NOT NULL	DATE
JOB_ID	NOT NULL	VARCHAR2(10)
DEPARTMENT_ID		NUMBER(4)

```
SELECT * FROM job_history;
```

EMPLOYEE_ID	START_DAT	END_DATE	JOB_ID	deptid
102	13-JAN-93	24-JUL-98	IT_PROG	60
101	21-SEP-89	27-OCT-93	AC_ACCOUNT	110
101	28-OCT-93	15-MAR-97	AC_MGR	110
201	17-FEB-96	19-DEC-99	MK_REP	20
114	24-MAR-98	31-DEC-99	ST_CLERK	50
122	01-JAN-99	31-DEC-99	ST_CLERK	50
200	17-SEP-87	17-JUN-93	AD_ASST	90
176	24-MAR-98	31-DEC-98	SA_REP	80
176	01-JAN-99	31-DEC-99	SA_MAN	80
200	01-JUL-94	31-DEC-98	AC_ACCOUNT	90

10 rows selected.

`REF` Cursors

Cursor Variables

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

Cursor Variables

Cursor variables are like C or Pascal pointers, which hold the memory location (address) of some item instead of the item itself. Thus, declaring a cursor variable creates a pointer, not an item. In PL/SQL, a pointer has the data type `REF X`, where `REF` is short for `REFERENCE` and `X` stands for a class of objects. A cursor variable has the `REF CURSOR` data type.

Like a cursor, a cursor variable points to the current row in the result set of a multirow query. However, cursors differ from cursor variables the way constants differ from variables. A cursor is static, but a cursor variable is dynamic because it is not tied to a specific query. You can open a cursor variable for any type-compatible query. This gives you more flexibility.

Cursor variables are available to every PL/SQL client. For example, you can declare a cursor variable in a PL/SQL host environment such as an OCI or Pro*C program, and then pass it as an input host variable (bind variable) to PL/SQL. Moreover, application development tools such as Oracle Forms and Oracle Reports, which have a PL/SQL engine, can use cursor variables entirely on the client side. The Oracle server also has a PL/SQL engine. You can pass cursor variables back and forth between an application and server through remote procedure calls (RPCs).

Why Use Cursor Variables?

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

ORACLE

Why Use Cursor Variables?

You use cursor variables to pass 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 in which the result set is stored. For example, an OCI client, an Oracle Forms application, and the Oracle server can all refer to the same work area.

A query work area remains accessible as long as any cursor variable points to it. Therefore, you can pass the value of a cursor variable freely from one scope to another. For example, if you pass a host cursor variable to a PL/SQL block that is embedded in a Pro*C program, the work area to which the cursor variable points remains accessible after the block completes.

If you have a PL/SQL engine on the client side, calls from client to server impose no restrictions. For example, you can declare a cursor variable on the client side, open and fetch from it on the server side, then continue to fetch from it back on the client side. Also, you can reduce network traffic by having a PL/SQL block open (or close) several host cursor variables in a single round trip.

A cursor variable holds a reference to the cursor work area in the PGA instead of addressing it with a static name. Because you address this area by a reference, you gain the flexibility of a variable.

Defining REF CURSOR Types

Define a REF CURSOR type:

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

Declare a cursor variable of that type:

```
ref_cv ref_type_name;
```

Example

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


Defining REF CURSOR Types

To define a REF CURSOR, you perform two steps. First, you define a REF CURSOR type, and then you declare cursor variables of that type. You can define REF CURSOR types in any PL/SQL block, subprogram, or package using the following syntax:

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

in which:

ref_type_name	Is a type specifier used in subsequent declarations of cursor variables
return_type	Represents a record or a row in a database table

In the above example, you specify a return type that represents a row in the database table DEPARTMENT.

REF CURSOR types can be strong (restrictive) or weak (nonrestrictive). As the next example shows, a strong REF CURSOR type definition specifies a return type, but a weak definition does not:

```
DECLARE
   TYPE EmpCurTyp IS REF CURSOR RETURN employees%ROWTYPE;   --
strong
   TYPE GenericCurTyp IS REF CURSOR;  -- weak
```

Defining REF CURSOR Types (continued)

Strong REF CURSOR types are less error prone because the PL/SQL compiler lets you associate a strongly typed cursor variable only with type-compatible queries. However, weak REF CURSOR types are more flexible because the compiler lets you associate a weakly typed cursor variable with any query.

Declaring Cursor Variables

After you define a REF CURSOR type, you can declare cursor variables of that type in any PL/SQL block or subprogram. In the following example, you declare the cursor variable DEPT_CV:

```
DECLARE
   TYPE DeptCurTyp IS REF CURSOR RETURN departments%ROWTYPE;
   dept_cv DeptCurTyp;  -- declare cursor variable
```

Note: You cannot declare cursor variables in a package. Unlike packaged variables, cursor variables do not have persistent states. Remember, declaring a cursor variable creates a pointer, not an item. Cursor variables cannot be saved in the database; they follow the usual scoping and instantiation rules.

In the RETURN clause of a REF CURSOR type definition, you can use %ROWTYPE to specify a record type that represents a row returned by a strongly (not weakly) typed cursor variable, as follows:

```
DECLARE
   TYPE TmpCurTyp IS REF CURSOR RETURN employees%ROWTYPE;
   tmp_cv TmpCurTyp;  -- declare cursor variable
   TYPE EmpCurTyp IS REF CURSOR RETURN tmp_cv%ROWTYPE;
   emp_cv EmpCurTyp;  -- declare cursor variable
```

Likewise, you can use %TYPE to provide the data type of a record variable, as the following example shows:

```
DECLARE
   dept_rec departments%ROWTYPE;  -- declare record variable
   TYPE DeptCurTyp IS REF CURSOR RETURN dept_rec%TYPE;
   dept_cv DeptCurTyp;  -- declare cursor variable
```

In the final example, you specify a user-defined RECORD type in the RETURN clause:

```
DECLARE
   TYPE EmpRecTyp IS RECORD (
      empno NUMBER(4),
      ename VARCHAR2(10),
      sal   NUMBER(7,2));
   TYPE EmpCurTyp IS REF CURSOR RETURN EmpRecTyp;
   emp_cv EmpCurTyp;  -- declare cursor variable
```

Cursor Variables As Parameters

You can declare cursor variables as the formal parameters of functions and procedures. In the following example, you define the `REF CURSOR` type `EmpCurTyp`, and then declare a cursor variable of that type as the formal parameter of a procedure:

```
DECLARE
   TYPE EmpCurTyp IS REF CURSOR RETURN emp%ROWTYPE;
   PROCEDURE open_emp_cv (emp_cv IN OUT EmpCurTyp) IS ...
```

Using the OPEN-FOR, FETCH, and CLOSE Statements

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

Using the OPEN-FOR, FETCH, and CLOSE Statements

You use three statements to process a dynamic multirow query: OPEN-FOR, FETCH, and CLOSE. First, you "open" a cursor variable "for" a multirow query. Then, you "fetch" rows from the result set one at a time. When all the rows are processed, you "close" the cursor variable.

Opening the Cursor Variable

The OPEN-FOR statement associates a cursor variable with a multirow query, executes the query, identifies the result set, positions the cursor to point to the first row of the results set, then sets the rows-processed count kept by %ROWCOUNT to zero. Unlike the static form of OPEN-FOR, the dynamic form has an optional USING clause. At run time, bind arguments in the USING clause replace corresponding placeholders in the dynamic SELECT statement. The syntax is:

```
OPEN {cursor_variable | :host_cursor_variable} FOR
dynamic_string
              [USING bind_argument[, bind_argument]...];
```

where CURSOR_VARIABLE is a weakly typed cursor variable (one without a return type), HOST_CURSOR_VARIABLE is a cursor variable declared in a PL/SQL host environment such as an OCI program, and dynamic_string is a string expression that represents a multirow query.

Using the `OPEN-FOR`, `FETCH`, and `CLOSE` Statements (continued)

In the following example, the syntax declares a cursor variable, and then associates it with a dynamic `SELECT` statement that returns rows from the `employees` table:

```
DECLARE
 TYPE EmpCurTyp IS REF CURSOR;  -- define weak REF CURSOR
   type
 emp_cv   EmpCurTyp;  -- declare cursor variable
 my_ename VARCHAR2(15);
 my_sal   NUMBER := 1000;
BEGIN
 OPEN emp_cv FOR  -- open cursor variable
    'SELECT last_name, salary FROM employees WHERE salary >
             :s'
     USING my_sal;
 ...
END;
```

Any bind arguments in the query are evaluated only when the cursor variable is opened. Thus, to fetch rows from the cursor using different bind values, you must reopen the cursor variable with the bind arguments set to their new values.

Fetching from the Cursor Variable

The `FETCH` statement returns a row from the result set of a multirow query, assigns the values of select-list items to corresponding variables or fields in the `INTO` clause, increments the count kept by `%ROWCOUNT`, and advances the cursor to the next row. Use the following syntax:

```
FETCH {cursor_variable | :host_cursor_variable}
 INTO {define_variable[, define_variable]... | record};
```

Continuing the example, fetch rows from cursor variable `EMP_CV` into define variables `MY_ENAME` and `MY_SAL`:

```
LOOP
 FETCH emp_cv INTO my_ename, my_sal;  -- fetch next row
 EXIT WHEN emp_cv%NOTFOUND;  -- exit loop when last row is
   fetched
 -- process row
END LOOP;
```

For each column value returned by the query associated with the cursor variable, there must be a corresponding, type-compatible variable or field in the `INTO` clause. You can use a different `INTO` clause on separate fetches with the same cursor variable. Each fetch retrieves another row from the same result set. If you try to fetch from a closed or never-opened cursor variable, PL/SQL raises the predefined exception `INVALID_CURSOR`.

Using the `OPEN-FOR`, `FETCH`, and `CLOSE` Statements (continued)

Closing the Cursor Variable

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

```
CLOSE {cursor_variable | :host_cursor_variable};
```

In this example, when the last row is processed, close the EMP_CV cursor variable:

```
LOOP
  FETCH emp_cv INTO my_ename, my_sal;
  EXIT WHEN emp_cv%NOTFOUND;
  -- process row
END LOOP;
CLOSE emp_cv;  -- close cursor variable
```

If you try to close an already-closed or never-opened cursor variable, PL/SQL raises INVALID_CURSOR.

An Example of Fetching

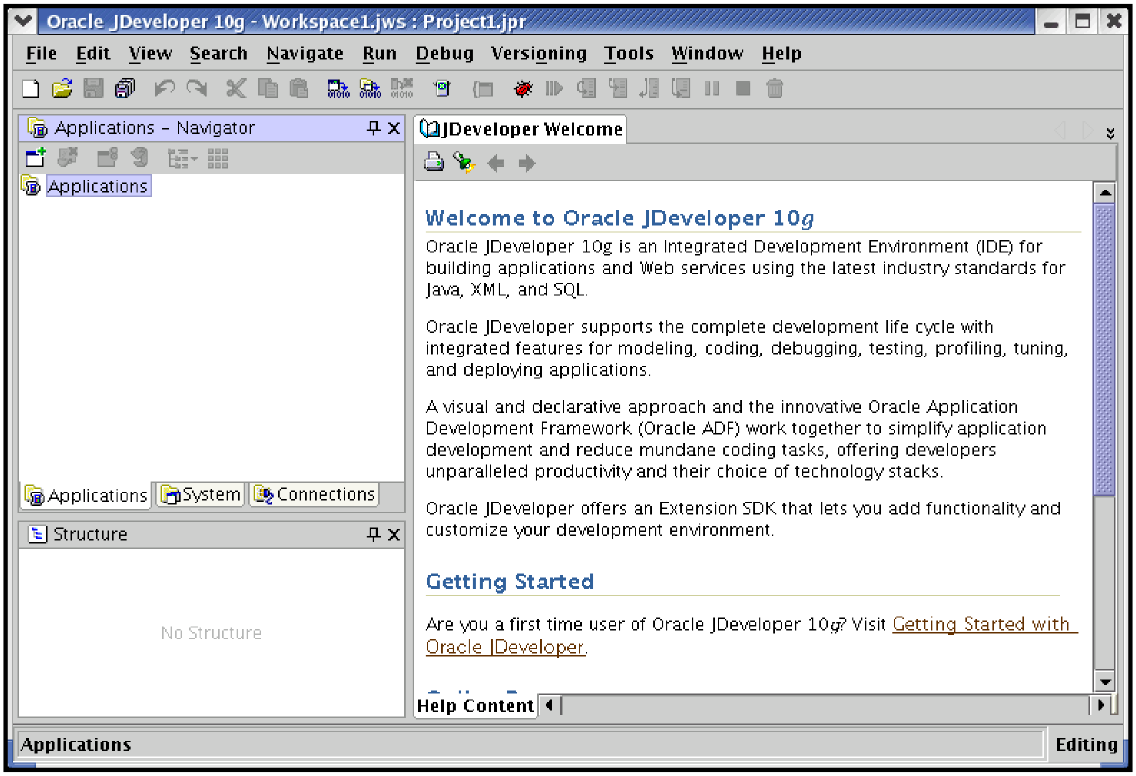
```
DECLARE
   TYPE EmpCurTyp IS REF CURSOR;
   emp_cv    EmpCurTyp;
   emp_rec   employees%ROWTYPE;
   sql_stmt VARCHAR2(200);
   my_job    VARCHAR2(10) := 'ST_CLERK';
BEGIN
   sql_stmt := 'SELECT * FROM employees
                WHERE job_id = :j';
   OPEN emp_cv FOR sql_stmt USING my_job;
   LOOP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


An Example of Fetching

The example in the slide shows that you can fetch rows from the result set of a dynamic multirow query into a record. First, you must define a `REF CURSOR` type, `EmpCurTyp`. Next, you define a cursor variable `emp_cv`, of the type `EmpcurTyp`. In the executable section of the PL/SQL block, the `OPEN-FOR` statement associates the cursor variable `EMP_CV` with the multirow query, `sql_stmt`. The `FETCH` statement returns a row from the result set of a multirow query and assigns the values of select-list items to `EMP_REC` in the `INTO` clause. When the last row is processed, close the `EMP_CV` cursor variable.

Oracle JDeveloper

Oracle JDeveloper 10*g*

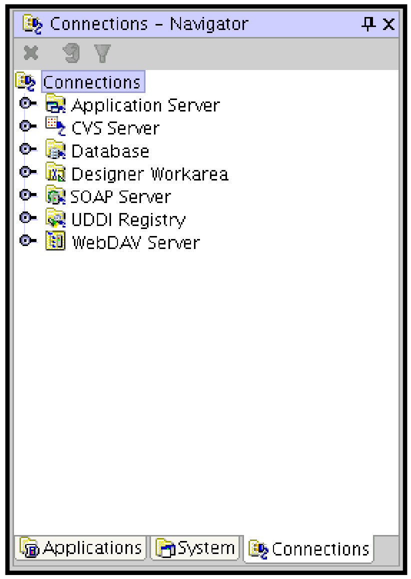


Oracle JDeveloper 10*g*

Oracle JDeveloper 10*g* is an integrated development environment (IDE) for developing and deploying Java applications and Web services. It supports every stage of the software development life cycle (SDLC) from modeling to deploying. It has the features to use the latest industry standards for Java, XML, and SQL while developing an application.

Oracle JDeveloper 10*g* initiates a new approach to J2EE development with the features that enables visual and declarative development. This innovative approach makes J2EE development simple and efficient.

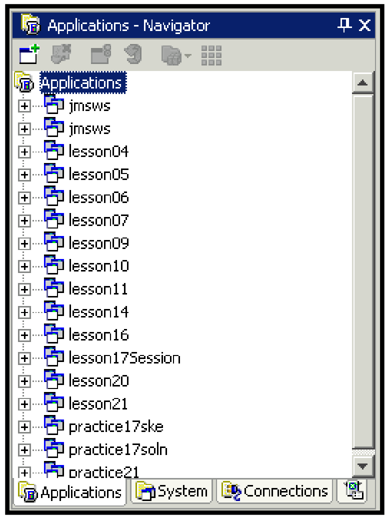
Connection Navigator



Connection Navigator

Using Oracle JDeveloper 10*g*, you can store the information necessary to connect to a database in an object called “connection.” A connection is stored as part of the IDE settings, and can be exported and imported for easy sharing among groups of users. A connection serves several purposes from browsing the database and building applications, all the way through to deployment.

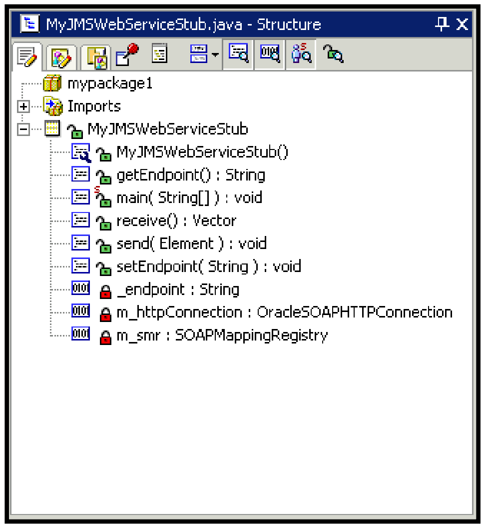
Application Navigator



Application Navigator

The Application Navigator gives you a logical view of your application and the data it contains. The Application Navigator provides an infrastructure that the different extensions can plug into and use to organize their data and menus in a consistent, abstract manner. While the Application Navigator can contain individual files (such as Java source files), it is designed to consolidate complex data. Complex data types such as entity objects, UML diagrams, EJB, or Web services appear in this navigator as single nodes. The raw files that make up these abstract nodes appear in the Structure window.

Structure Window



Structure Window

The Structure window offers a structural view of the data in the document currently selected in the active window of those windows that participate in providing structure: the navigators, the editors and viewers, and the Property Inspector.

In the Structure window, you can view the document data in a variety of ways. The structures available for display are based upon document type. For a Java file, you can view code structure, UI structure, or UI model data. For an XML file, you can view XML structure, design structure, or UI model data.

The Structure window is dynamic, tracking always the current selection of the active window (unless you freeze the window's contents on a particular view), as is pertinent to the currently active editor. When the current selection is a node in the navigator, the default editor is assumed. To change the view on the structure for the current selection, select a different structure tab.

Editor Window

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


Editor Window

You can view your project files all in one single editor window, you can open multiple views of the same file, or you can open multiple views of different files.

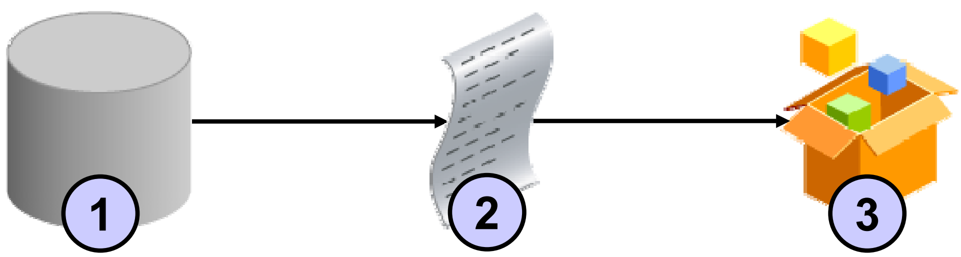
The tabs at the top of the editor window are the document tabs. Selecting a document tab gives that file focus, bringing it to the foreground of the window in the current editor.

The tabs at the bottom of the editor window for a given file are the editor tabs. Selecting an editor tab opens the file in that editor.

Deploying Java Stored Procedures

Before deploying Java stored procedures, perform the following steps:

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



Deploying Java Stored Procedures

Create a deployment profile for Java stored procedures, and then deploy the classes and, optionally, any public static methods in JDeveloper using the settings in the profile.

Deploying to the database uses the information provided in the Deployment Profile Wizard and two Oracle Database utilities:

- `loadjava` loads the Java class containing the stored procedures to an Oracle database.
- `publish` generates the PL/SQL call specific wrappers for the loaded public static methods. Publishing enables the Java methods to be called as PL/SQL functions or procedures.

Publishing Java to PL/SQL

FormatCreditCardNo.java | CCFORMAT

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

FormatCreditCardNo.java | CCFORMAT

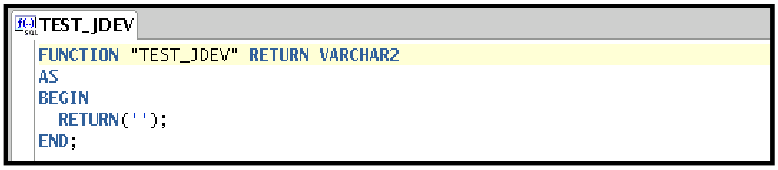
```sql
PROCEDURE ccformat (x IN OUT varchar2)
AS LANGUAGE JAVA
NAME 'FormatCreditCardNo.formatCard(java.lang.String[])';
```

ORACLE

Publishing Java to PL/SQL

The slide shows the Java code and how to publish the Java code in a PL/SQL procedure.

Creating Program Units



Skeleton of the function

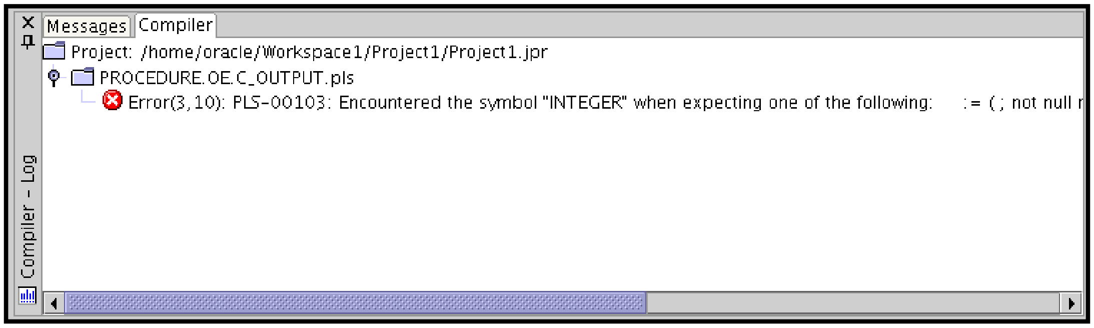
Creating Program Units

To create a PL/SQL program unit:

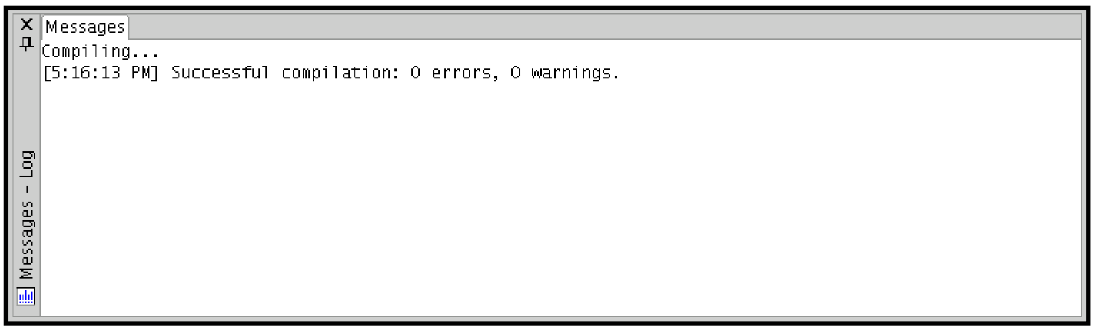
1. Select View > Connection Navigator.
2. Expand Database and select a database connection.
3. In the connection, expand a schema.
4. Right-click a folder corresponding to the object type (Procedures, Packages, Functions).
5. Choose New PL/SQL object_type. The Create PL/SQL dialog box appears for the function, package, or procedure.
6. Enter a valid name for the function, package, or procedure and click OK.

A skeleton definition will be created and opened in the Code Editor. You can then edit the subprogram to suit your need.

Compiling



Compilation with errors

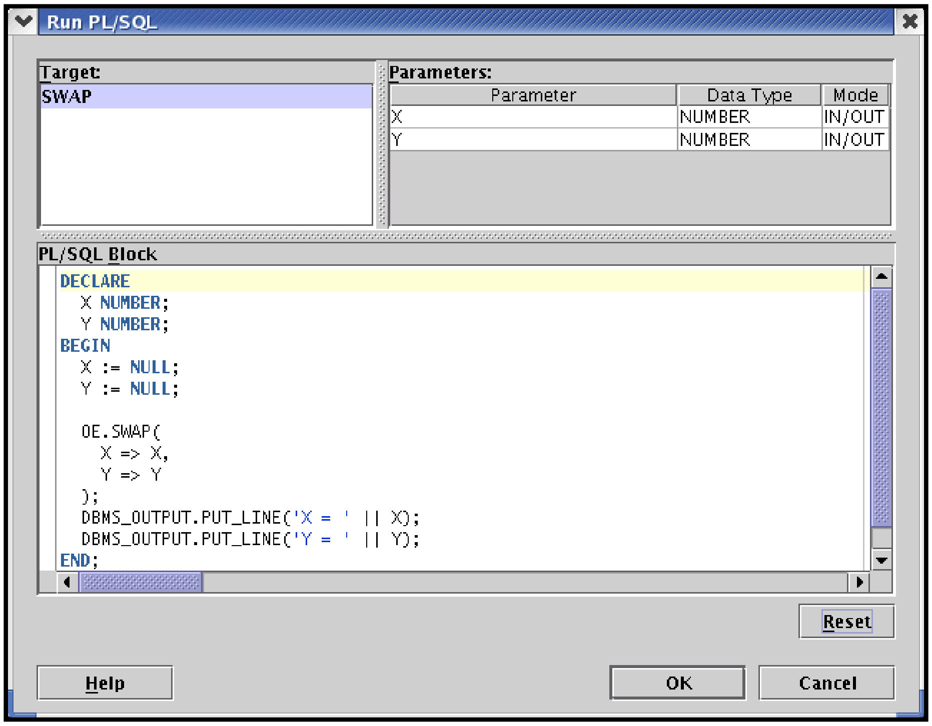


Compilation without errors

Compiling

After editing the skeleton definition, you need to compile the program unit. Right-click the PL/SQL object that you need to compile in the Connection Navigator and then select Compile. Alternatively you can also press CTRL + SHIFT + F9 to compile.

Running a Program Unit

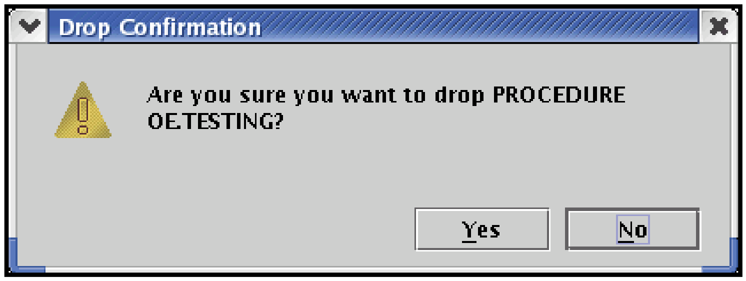


ORACLE

Running a Program Unit

To execute the program unit, right-click the object and click Run. The Run PL/SQL dialog box will appear. You may need to change the `NULL` values with reasonable values that are passed into the program unit. After you change the values, click OK. The output will be displayed in the Message-Log window.

Dropping a Program Unit



Dropping a Program Unit

To drop a program unit, right-click the object and select Drop. The Drop Confirmation dialog box will appear; click Yes. The object will be dropped from the database.

Debugging PL/SQL Programs

JDeveloper support two types of debugging:

- **Local**
- **Remote**

You need the following privileges to perform PL/SQL debugging:

- **DEBUG ANY PROCEDURE**
- **DEBUG CONNECT SESSION**

Debugging PL/SQL Programs

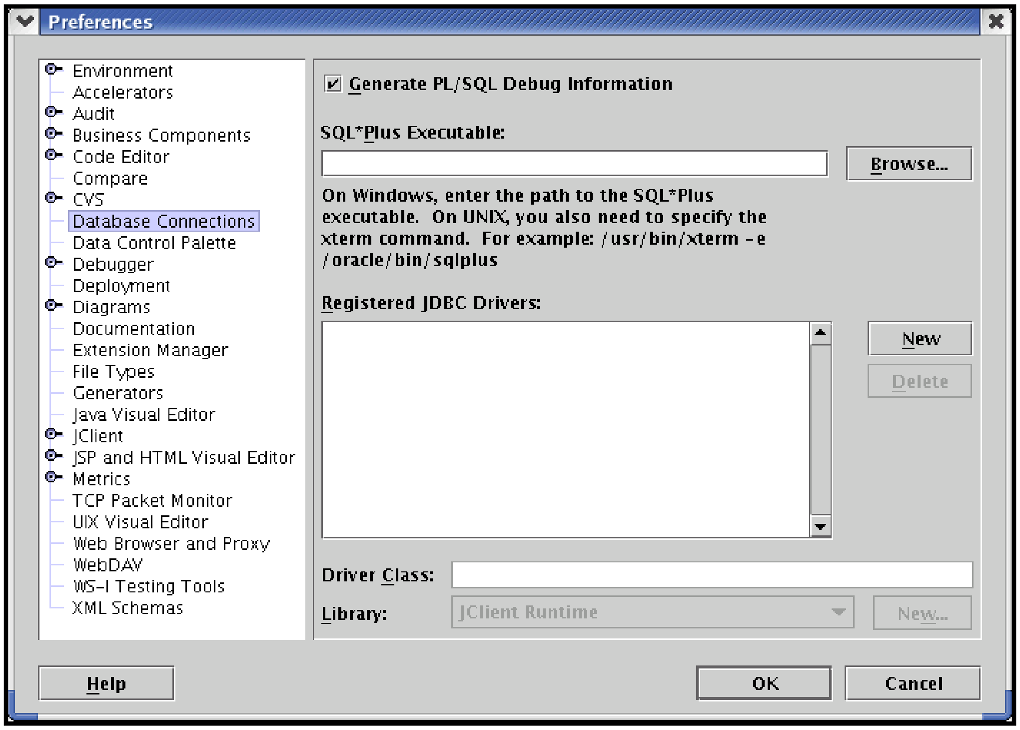
JDeveloper offers both local and remote debugging. A local debugging session is started by setting breakpoints in source files, and then starting the debugger. Remote debugging requires two JDeveloper processes: a `debugger` and a `debuggee` which may reside on a different platform.

To debug a PL/SQL program it must be compiled in `INTERPRETED` mode. You cannot debug a PL/SQL program that is compiled in `NATIVE` mode. This mode is set in the database's `init.ora` file.

PL/SQL programs must be compiled with the `DEBUG` option enabled. This option can be enabled using various ways. Using SQL*Plus, execute `ALTER SESSION SET PLSQL_DEBUG = true` to enable the `DEBUG` option. Then you can create or recompile the PL/SQL program you want to debug. Another way of enabling the `DEBUG` option is by using the following command in SQL*Plus:

```
ALTER <procedure, function, package> <name> COMPILE DEBUG;
```

Debugging PL/SQL Programs



Debugging PL/SQL Programs (continued)

Before you start with debugging, make sure that the Generate PL/SQL Debug Information check box is selected. You can access the dialog box by using Tools > Preferences > Database Connections.

Instead of manually testing PL/SQL functions and procedures as you may be accustomed to doing from within SQL*Plus or by running a dummy procedure in the database, JDeveloper enables you to test these objects in an automatic way. With this release of JDeveloper, you can run and debug PL/SQL program units. For example, you can specify parameters being passed or return values from a function giving you more control over what is run and providing you output details about what was tested.

Note: The procedures or functions in the Oracle database can be either stand-alone or within a package.

Debugging PL/SQL Programs (continued)

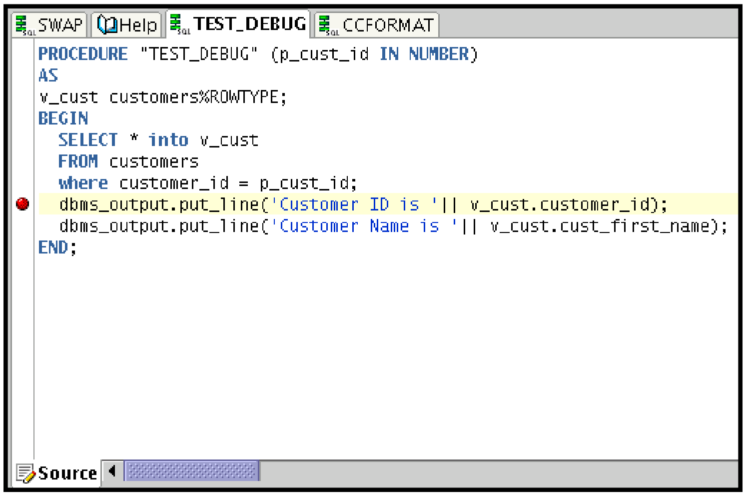
To run or debug functions, procedures, and packages:

1. Create a database connection using the Database Wizard.
2. In the Navigator, expand the Database node to display the specific database username and schema name.
3. Expand the Schema node.
4. Expand the appropriate node depending on what you are debugging: Procedure, Function, or Package body.
5. (Optional for debugging only) Select the function, procedure, or package that you want to debug and double-click to open it in the Code Editor.
6. (Optional for debugging only) Set a breakpoint in your PL/SQL code by clicking to the left of the margin.
   Note: The breakpoint must be set on an executable line of code. If the debugger does not stop, the breakpoint may have not been set on an executable line of code (verify that the breakpoint was set correctly). Also, verify that the debugging PL/SQL prerequisites were met. In particular, make sure that the PL/SQL program is compiled in the `INTERPRETED` mode.
7. Make sure that either the Code Editor or the procedure in the Navigator is currently selected.
8. Click the Debug toolbar button, or, if you want to run without debugging, click the Run toolbar button.
9. The Run PL/SQL dialog box is displayed.
   - Select a target that is the name of the procedure or function that you want to debug. Note that the content in the Parameters and PL/SQL Block boxes change dynamically when the target changes.

   Note: You will have a choice of target only if you choose to run or debug a package that contains more than one program unit.
   The Parameters box lists the target's arguments (if applicable).
   The PL/SQL Block box displays code that was custom generated by JDeveloper for the selected target. Depending on what the function or procedure does, you may need to replace the NULL values with reasonable values so that these are passed into the procedure, function, or package. In some cases, you may need to write additional code to initialize values to be passed as arguments. In this case, you can edit the PL/SQL block text as necessary.
10. Click OK to execute or debug the target.
11. Analyze the output information displayed in the Log window.

In the case of functions, the return value will be displayed. `DBMS_OUTPUT` messages will also be displayed.

Setting Breakpoints

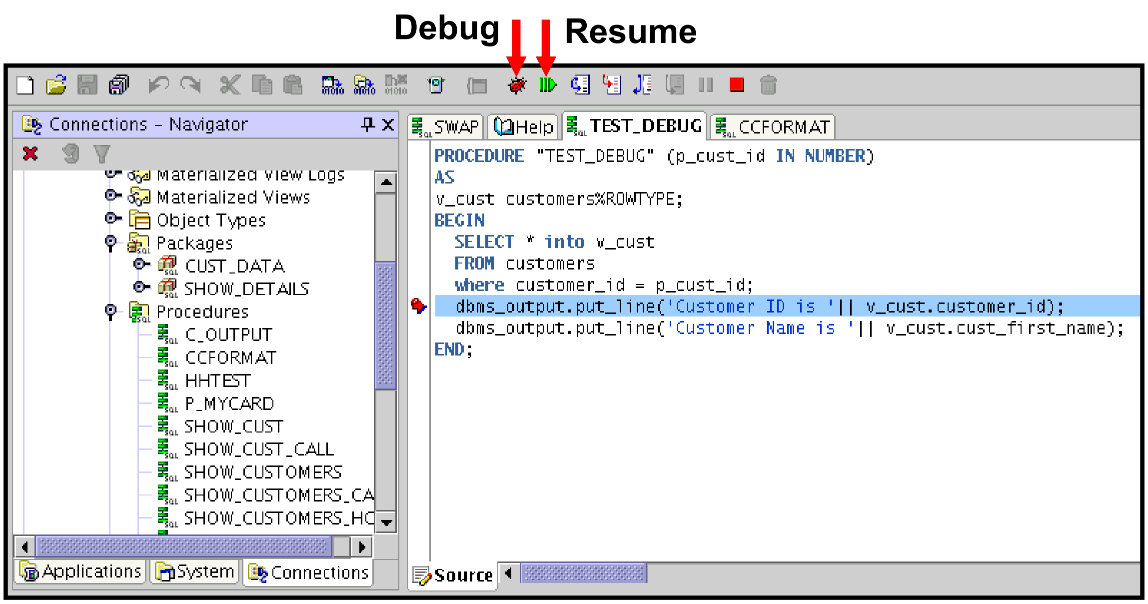


Setting Breakpoints

Breakpoints help you to examine the values of the variables in your program. It is a trigger in a program that, when reached, pauses program execution allowing you to examine the values of some or all of the program variables. By setting breakpoints in potential problem areas of your source code, you can run your program until its execution reaches a location you want to debug. When your program execution encounters a breakpoint, the program pauses, and the debugger displays the line containing the breakpoint in the Code Editor. You can then use the debugger to view the state of your program. Breakpoints are flexible in that they can be set before you begin a program run or at any time while you are debugging.

To set a breakpoint in the code editor, click the left margin next to a line of executable code. Breakpoints set on comment lines, blank lines, declaration and any other non-executable lines of code are not verified by the debugger and are treated as invalid.

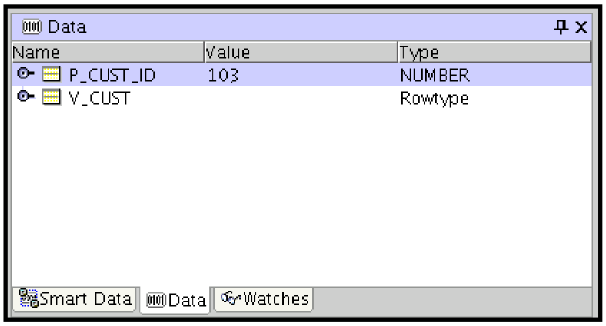
Stepping Through Code



Stepping Through Code

After setting the breakpoint, start the debugger by clicking the Debug icon. The debugger will pause the program execution at the point where the breakpoint is set. At this point, you can check the values of the variables. You can continue with the program execution by clicking the Resume icon. The debugger will then move on to the next breakpoint. After executing all the breakpoints, the debugger will stop the execution of the program and display the results in the Debugging – Log area.

Examining and Modifying Variables



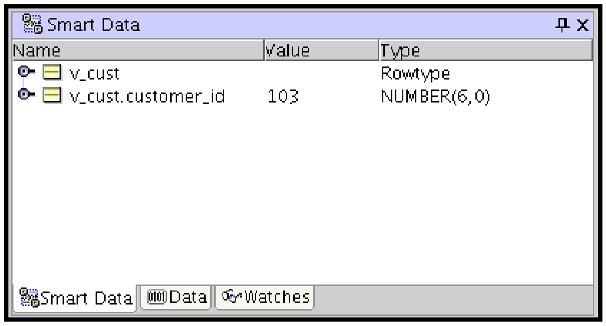
Data window

Examining and Modifying Variables

When the debugging is ON, you can examine and modify the value of the variables using the Data, Smart Data, and Watches windows. You can modify program data values during a debugging session as a way to test hypothetical bug fixes during a program run. If you find that a modification fixes a program error, you can exit the debugging session, fix your program code accordingly, and recompile the program to make the fix permanent.

You use the Data window to display information about variables in your program. The Data window displays the arguments, local variables, and static fields for the current context, which is controlled by the selection in the Stack window. If you move to a new context, the Data window is updated to show the data for the new context. If the current program was compiled without debug information, you will not be able to see the local variables.

Examining and Modifying Variables

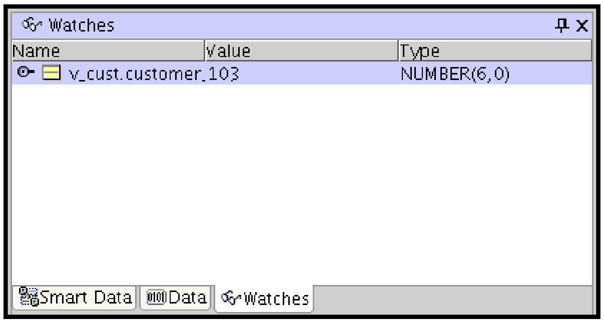


Smart Data window

Examining and Modifying Variables (continued)

Unlike the Data window that displays all the variables in your program, the Smart Data window displays only the data that is relevant to the source code that you are stepping through.

Examining and Modifying Variables

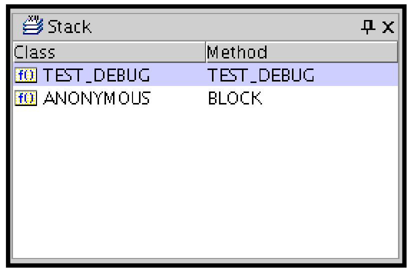


Watches window

Examining and Modifying Variables (continued)

A watch enables you to monitor the changing values of variables or expressions as your program runs. After you enter a watch expression, the Watch window displays the current value of the expression. As your program runs, the value of the watch changes as your program updates the values of the variables in the watch expression.

Examining and Modifying Variables

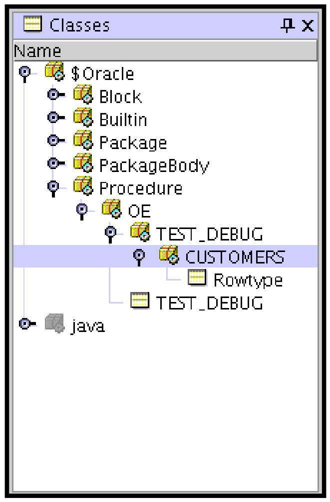


Stack window

Examining and Modifying Variables (continued)

You can activate the Stack window by using View > Debugger > Stack. It displays the call stack for the current thread. When you select a line in the Stack window, the Data window, Watch window, and all other windows are updated to show data for the selected class.

Examining and Modifying Variables



Classes window

Examining and Modifying Variables (continued)

The Classes window displays all the classes that are currently being loaded to execute the program. If used with Oracle Java Virtual Machine (OJVM), it also shows the number of instances of a class and the memory used by those instances.

E

Using SQL Developer

Objectives

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

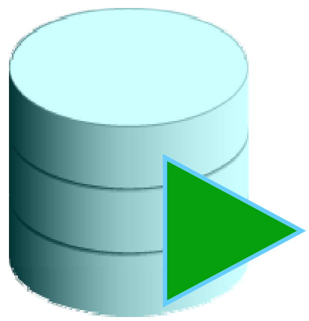
- List the key features of Oracle SQL Developer
- Install Oracle SQL Developer
- Identify menu items of Oracle SQL Developer
- Create a database connection
- Manage database objects
- Use the SQL Worksheet
- Execute SQL statements and SQL scripts
- Edit and Debug PL/SQL statements
- Create and save reports

Objectives

This appendix introduces the graphical tool SQL Developer that simplifies your database development tasks. You learn how to use SQL Worksheet to execute SQL statements and SQL scripts. You also learn how to edit and debug PL/SQL.

What Is Oracle SQL Developer?

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



SQL Developer

ORACLE

What Is Oracle SQL Developer?

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

SQL Developer, the visual tool for database development, 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. Once connected, you can perform operations on objects in the database.

Key Features

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

ORACLE

Key Features of SQL Developer

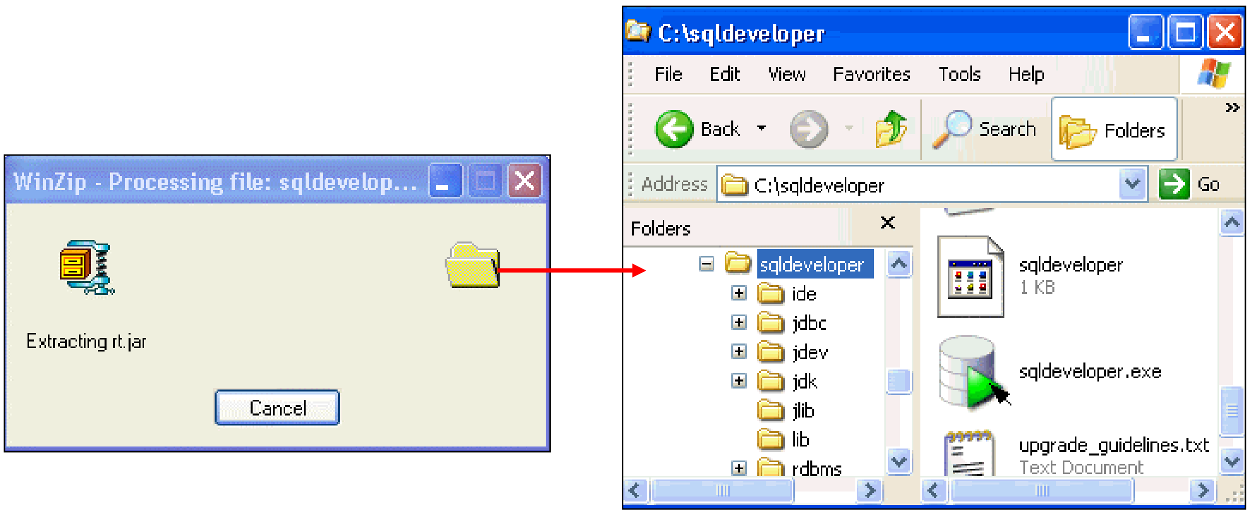
Oracle SQL Developer is developed in Java leveraging the Oracle JDeveloper IDE. The tool runs on Windows, Linux, and Mac OS X platforms. You can install SQL Developer on the Database Server and connect remotely from your desktop, thus avoiding client server network traffic.

Default connectivity to the database is through the JDBC Thin driver so, no Oracle Home is required. SQL Developer does not require an installer and you need to simply unzip the downloaded file.

With SQL Developer, users can connect to Oracle Databases 9.2.0.1 and later, and all Oracle database editions including Express Edition. SQL Developer is bundled with JRE 1.5, with an additional tools.jar to support Windows clients. Non-Windows clients only need JDK 1.5.

Installing SQL Developer

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



Installing SQL Developer

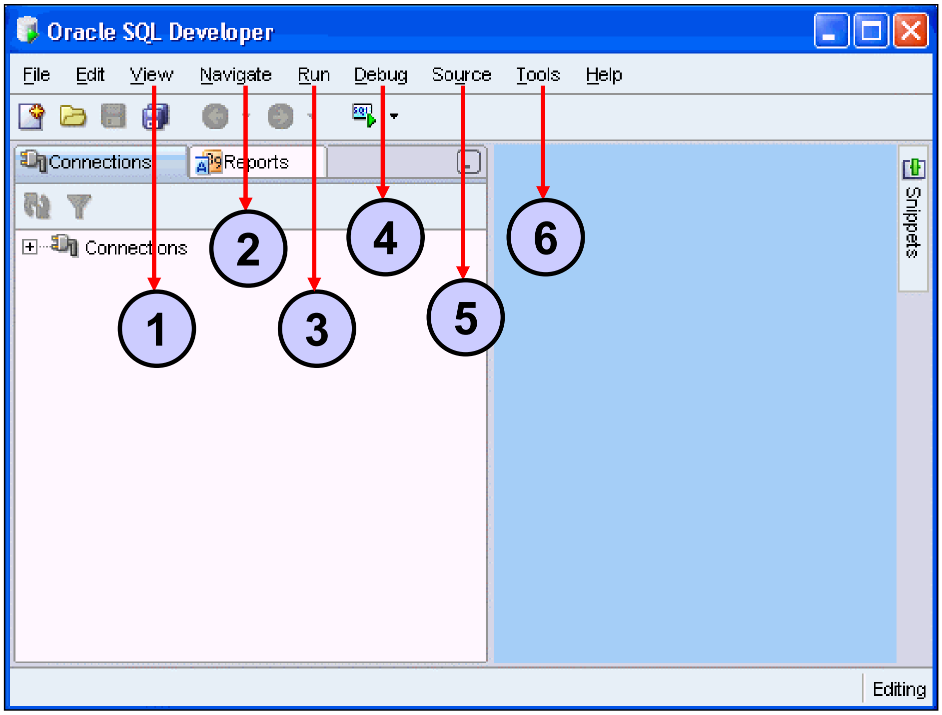
Oracle SQL Developer does not require an installer. To install SQL Developer, you need an unzip tool.

To install SQL Developer, perform the following steps:

1. Create a folder as **`<local drive>:\SQL Developer`**.
2. Download the SQL Developer kit from http://www.oracle.com/technology/software/products/sql/index.html
3. Unzip the downloaded SQL Developer kit into the folder created in step 1.

To start SQL Developer, go to **`<local drive>:\SQL Developer`**, and double-click **`sqldeveloper.exe`**.

Menus for SQL Developer



Menus for SQL Developer

SQL Developer has two main navigation tabs.

- **Connections Navigator:** By using this tab, you can browse database objects and users to which you have access.
- **Reporting Tab:** By using this tab, you can run predefined reports or create and add your own reports.

SQL Developer uses the left side for navigation to find and select objects, and the right side to display information about selected objects. You can customize many aspects of the appearance and behavior of SQL Developer by setting preferences.

The menus at the top contain standard entries, plus entries for features specific to SQL Developer.

1. **View:** Contains options that affect what is displayed in the SQL Developer interface
2. **Navigate:** Contains options for navigating to panes and in the execution of sub programs
3. **Run:** Contains the Run File and Execution Profile options, which are relevant when a function or procedure is selected
4. **Debug:** Contains options relevant when a function or procedure is selected
5. **Source:** Contains options for use when editing functions and procedures
6. **Tools:** Invokes SQL Developer tools such as SQL*Plus, Preferences, and SQL Worksheet

Creating a Database Connection

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

ORACLE

Creating a Database Connection

A connection is a SQL Developer object that specifies the necessary information for connecting to a specific database as a specific user of that database. To use SQL Developer, you must have at least one database connection, which may be existing, created, or imported.

You can create and test connections for multiple databases and for multiple schemas.

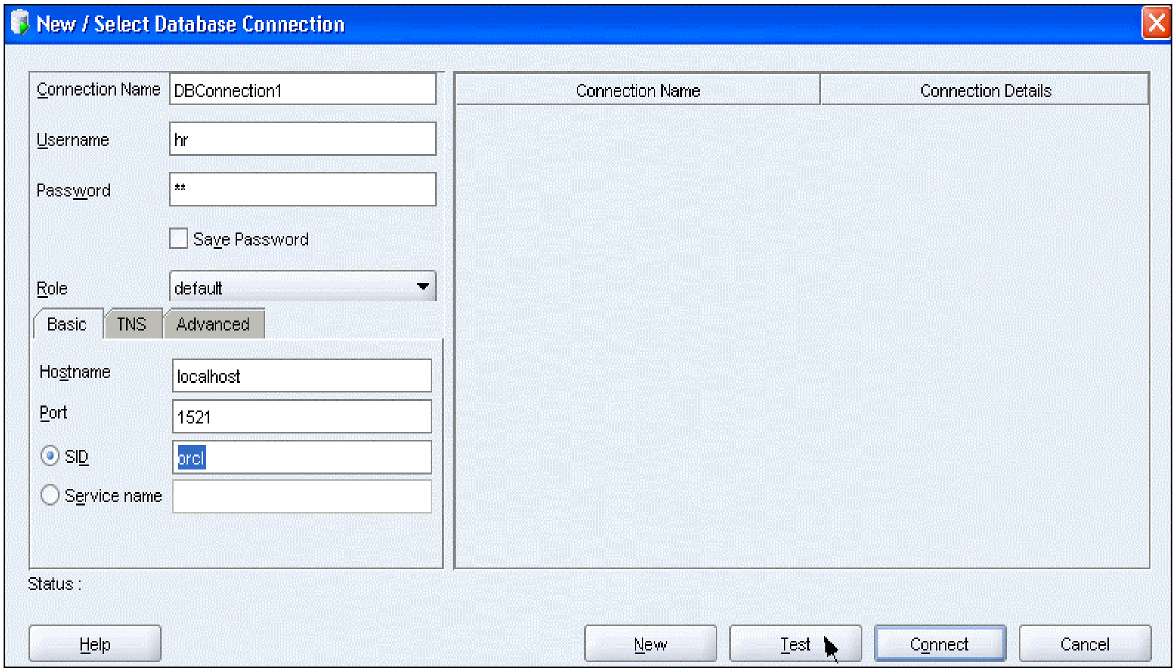
By default, the `tnsnames.ora` file is located in the `$ORACLE_HOME/network/admin directory`. But, it can also be in the directory specified by the `TNS_ADMIN` environment variable or registry value. When you start SQL Developer and display the database connections dialog box, SQL Developer automatically imports any connections defined in the tnsnames.ora file on your system.

Note: On Windows systems, if the tnsnames.ora file exists but its connections are not being used by SQL Developer, define `TNS_ADMIN` as a system environment variable.

You can export connections to an XML file so that you can reuse later.

You can create additional connections as different users to the same database or to connect to the different databases.

Creating a Database Connection



Creating a Database Connection (continued)

To create a database connection, perform the following steps:

1. Double-click **<your_path>\sqldeveloper\sqldeveloper.exe**.
2. In the Connections tab, right-click **Connections** and select **New Database Connection**.
3. Enter the connection name, username, password, hostname, and SID for the database you want to connect.
4. Click **Test** to make sure that the connection has been set correctly.
5. Click **Connect**.

   In the basic tabbed page, at the bottom, fill in the following options:

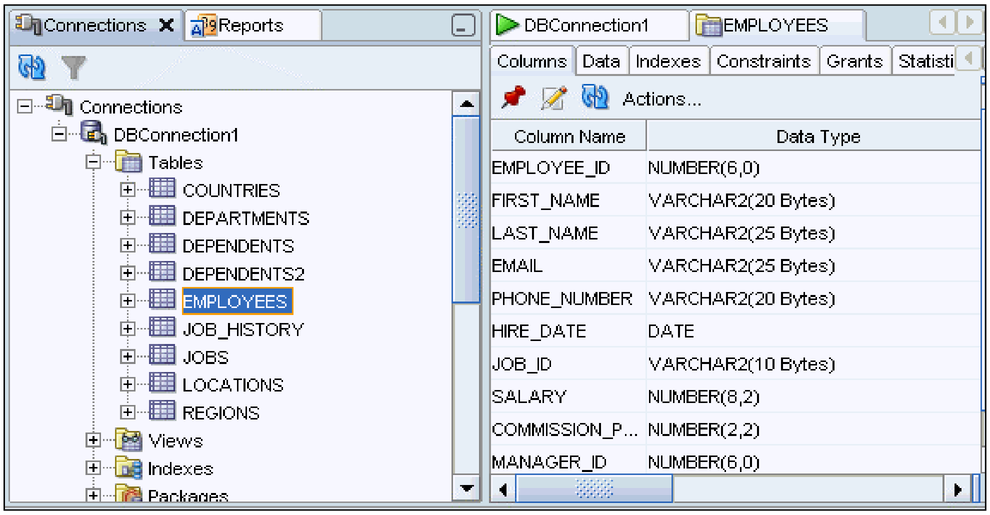
- **Hostname:** the Host system for the Oracle database
- **Port:** Listener port
- **SID:** Database name
- **Service Name:** Network service name for a remote database connection

If you select the Save Password check box, the password is saved to an XML file. So, once you close SQL Developer connection and open again, you will not be prompted for the password.

Browsing Database Objects

Use the Database Navigator to:

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



ORACLE

Browsing Database Objects

Once you have created a database connection, you can use the Database Navigator to browse through many objects in a database schema including Tables, Views, Indexes, Packages, Procedures, Triggers, Types, and so on.

SQL Developer uses the left side for navigation to find and select objects, and the right side to display information about the selected objects. You can customize many aspects of the appearance of SQL Developer by setting preferences.

You can see the definition of the objects broken into tabs of information that is pulled out of the data dictionary. For example, if you select a table in the Navigator, the details about columns, constraints, grants, statistics, triggers and more are all displayed in an easy to read tabbed window.

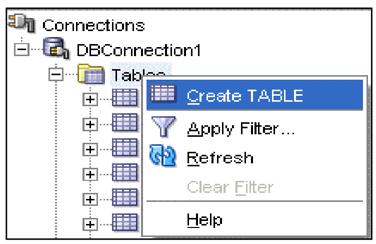
If you want to see the definition of `EMPLOYEES` table as shown on the slide, perform the following steps:

1. Expand the connection node in the Connections Navigator
2. Expand **Tables**.
3. Double-click **EMPLOYEES**.

Using the Data tab, you can enter new rows, update data and commit these changes to the database.

Creating a Schema Object

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



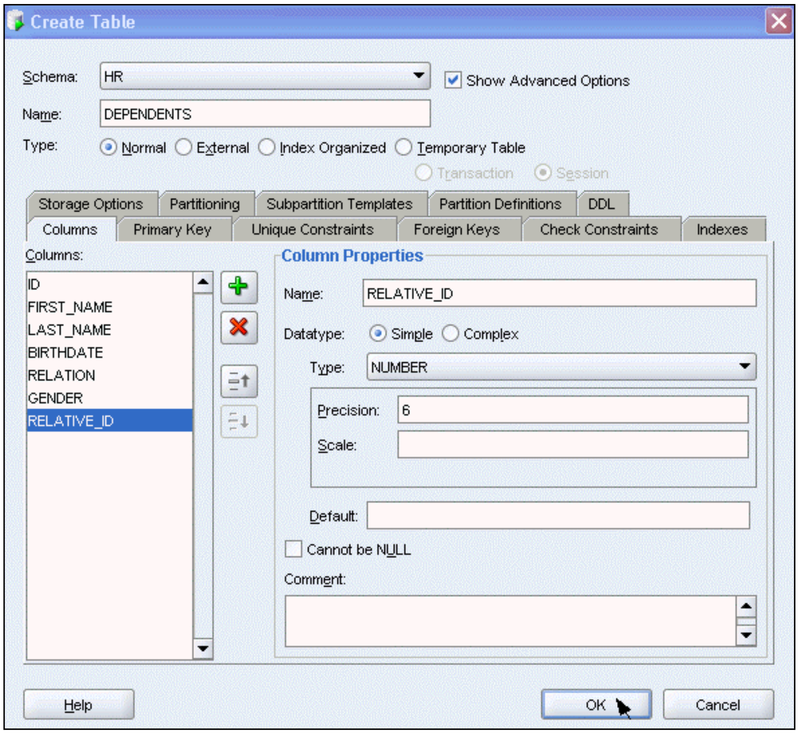
Creating a Schema Object

SQL Developer supports the creation of any schema object by executing a SQL statement in the SQL Worksheet. Alternatively, you can create objects using the context menus. Once created, you can edit the objects using an edit dialog or one of many context sensitive menus.

As new objects are created or existing objects are edited, the DDL for those adjustments is available for review. An Export DDL option is available if you want to create the full DDL for one or more objects in the schema.

The slide shows creating a table using the context menu. To open a dialog box for creating a new table, right-click **Tables** and select **Create TABLE**. The dialog boxes for creating and editing database objects have multiple tabs, each reflecting a logical grouping of properties for that type of object.

Creating a New Table: Example



Creating a New Table: Example

In the Create Table dialog box, if you do not select the **Show Advanced Options** check box, you can create a table quickly by specifying columns and some frequently used features.

If you select the **Show Advanced Options** check box, the Create Table dialog box changes to one with multiple tabs, in which you can specify an extended set of features while creating the table.

The example in the slide shows creating the `DEPENDENTS` table by selecting the **Show Advanced Options** check box.

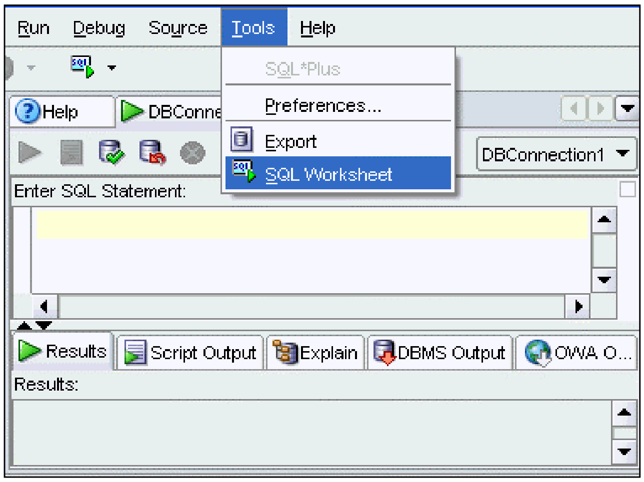
To create a new table, perform the following steps:

1. In the Connections Navigator, right-click **Tables**.
2. Select **Create TABLE**.
3. In the Create Table dialog box, select **Show Advanced Options**.
4. Specify column information.
5. Click **OK**.

Although it is not required, you should also specify a primary key using the Primary Key tab in the dialog box. Sometimes, you may want to edit the table that you have created. To edit a table, right-click the table in the connections navigator, and select **Edit**.

Using the SQL Worksheet

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



Using the SQL Worksheet

When you connect to a database, a SQL Worksheet window for that connection is automatically opened. You can use the SQL Worksheet to enter and execute SQL, PL/SQL, and SQL*Plus statements. The SQL Worksheet supports SQL*Plus statements to a certain extent. SQL*Plus statements that are not supported by the SQL Worksheet are ignored and not passed to the database.

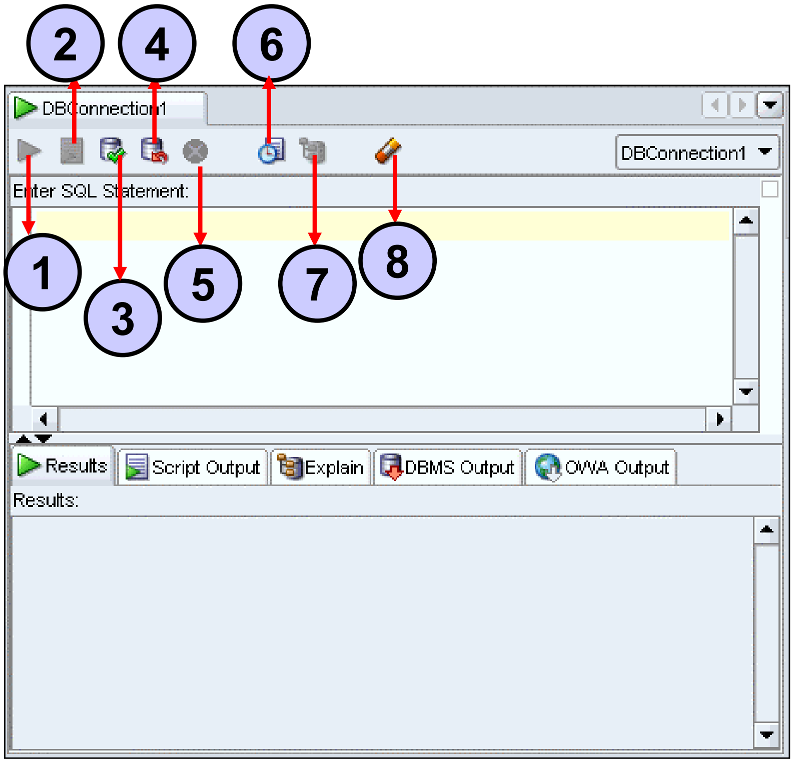
You can specify any actions that can be processed by the database connection associated with the worksheet, such as:

- Creating a table
- Inserting data
- Creating and editing a trigger
- Selecting data from a table
- Saving the selected data to a file

You can display a SQL Worksheet by using any of the following two options:

- Select **Tools > SQL Worksheet**
- Click the **Open SQL Worksheet** icon.

Using the SQL Worksheet



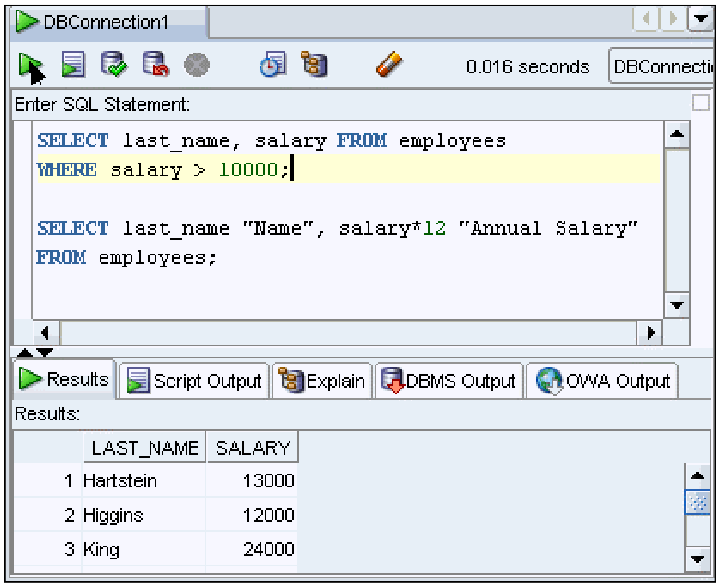
Using the SQL Worksheet (continued)

You may want to use short cut keys or icons to perform certain tasks such as executing a SQL statement, running a script, and viewing the history of SQL statements that you have executed. You can use the SQL Worksheet toolbar that contains icons to perform the following tasks:

1. **Execute Statement:** Executes the statement at the mouse pointer in the Enter SQL Statement box. You can use bind variables in the SQL statements but not substitution variables.
2. **Run Script:** Executes all statements in the Enter SQL Statement box using the Script Runner. You can use substitution variables in the SQL statements but not bind variables.
3. **Commit:** Writes any changes to the database, and ends the transaction.
4. **Rollback:** Discards any changes to the database, without writing them to the database, and ends the transaction.
5. **Cancel:** Stops the execution of any statements currently being executed.
6. **SQL History:** Displays a dialog box with information about SQL statements that you have executed.
7. **Execute Explain Plan:** Generates the execution plan, which you can see by clicking the Explain tab.
8. **Clear:** Erases the statement or statements in the Enter SQL Statement box.

Executing SQL Statements

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



Executing SQL Statements

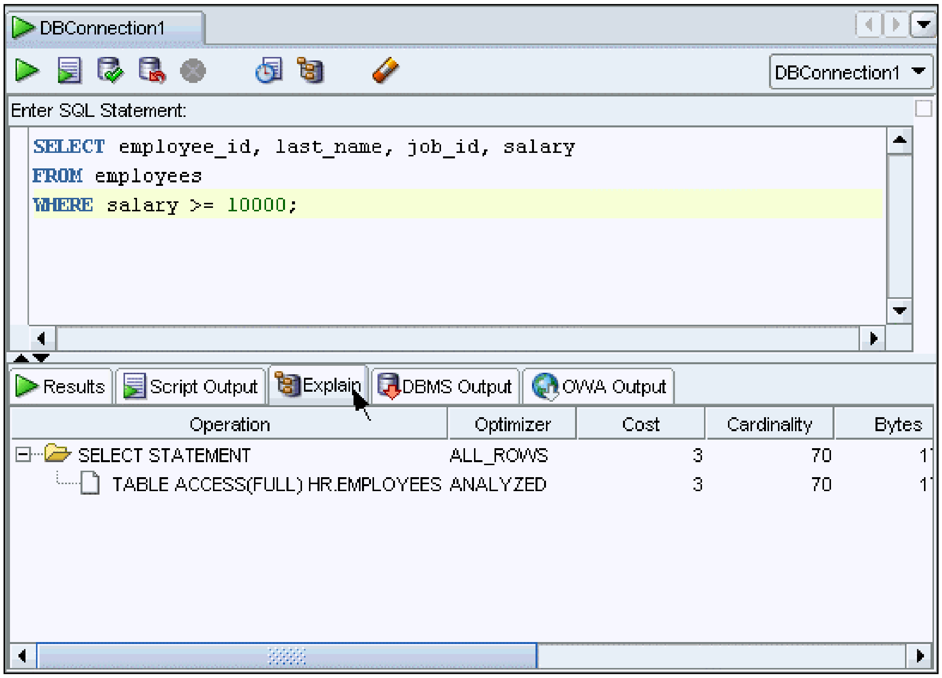
In the SQL Worksheet, you can use the Enter SQL Statement box to type a single or multiple SQL statements. For a single statement, the semicolon at the end is optional.

When you type in the statement, the SQL keywords are automatically highlighted. To execute a SQL statement, ensure that your cursor is within the statement and click the **Execute Statement** icon. Alternatively, you can press the **F9** key.

To execute multiple SQL statements and see the results, click the **Run Script** icon. Alternatively, you can press the **F5** key.

In the example in the slide, as there are multiple SQL statements, the first statement is terminated with a semicolon. The cursor is in the first statement and so when the statement is executed, results corresponding to the first statement are displayed in the Results box.

Viewing the Execution Plan



Viewing the Execution Plan

You can execute a SQL script, and view the execution plan. To execute a SQL script file, perform the following steps:

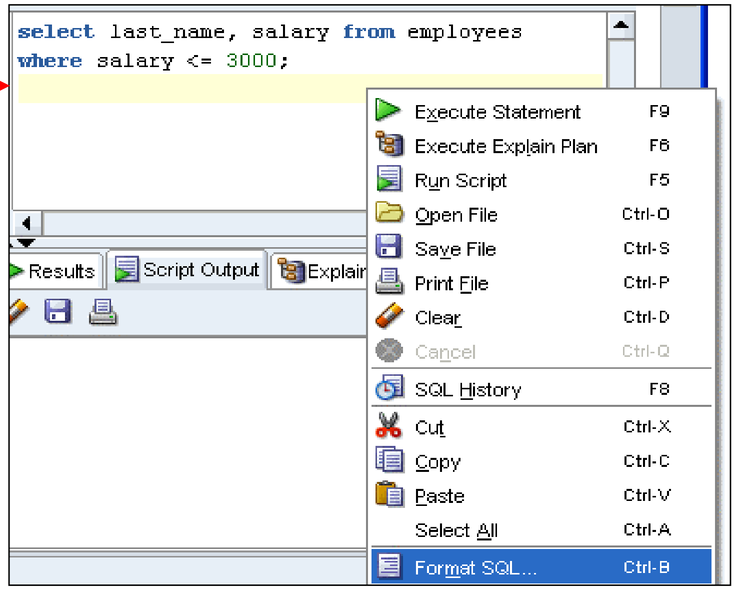
1. Right-click in the Enter SQL Statement box, and select **Open File** from the drop down menu.
2. In the Open dialog box, double-click the **.sql** file.
3. Click the **Run Script** icon.

Once you double-click the `.sql` file, the sql statements are loaded into the Enter SQL Statement box. You can execute the script or each line individually. The results are displayed in the Script Output area.

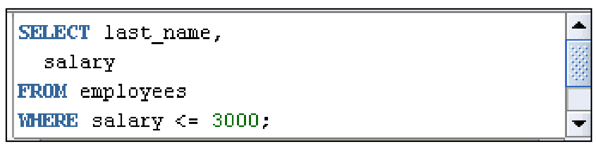
The example in the slide shows the execution plan. The Execute Explain Plan icon generates the execution plan. An execution plan is the sequence of operations that will be performed to execute the statement. You can see the execution plan by clicking the **Explain** tab.

Formatting SQL Code

Before formatting



After formatting



Formatting SQL Code

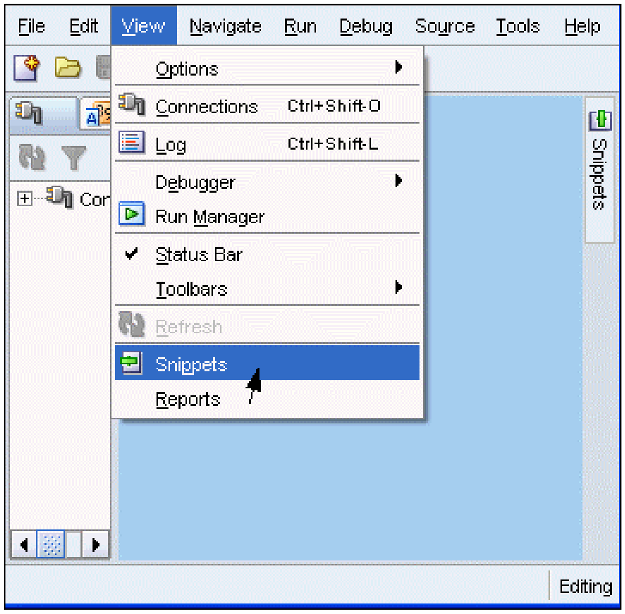
You may want to enhance the indentation, spacing, capitalization, and line separation of SQL code. SQL Developer enables you to format SQL code.

To format SQL code, right-click in the statement area, and select **Format SQL**.

In the example in the slide, before formatting, the SQL code has the key words not capitalized and the statement is not properly indented. After formatting, the SQL code is enhanced with the keywords capitalized and the statement properly indented.

Using Snippets

Snippets are code fragments that may be just syntax or examples



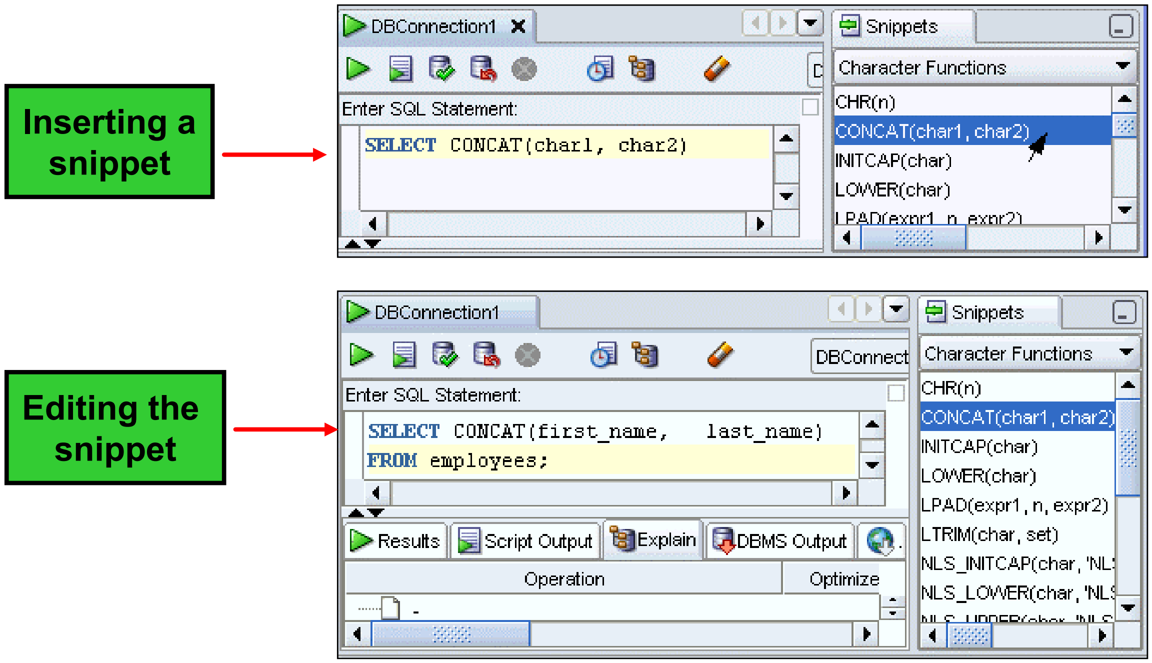
Using Snippets

You may want to use certain code fragments when you are using the SQL Worksheet or creating or editing a PL/SQL function or procedure. SQL Developer has the Snippets feature. Snippets are code fragments, such as SQL functions, Optimizer hints, and miscellaneous PL/SQL programming techniques. You can drag and drop snippets into the editor window.

To display Snippets, select **View > Snippets**.

The snippets window is displayed on the right side. You can use the drop down list to select a group. A snippets button is placed in the right window margin, so that you can display the snippets window if it becomes hidden.

Using Snippets: Example



Using Snippets: Example

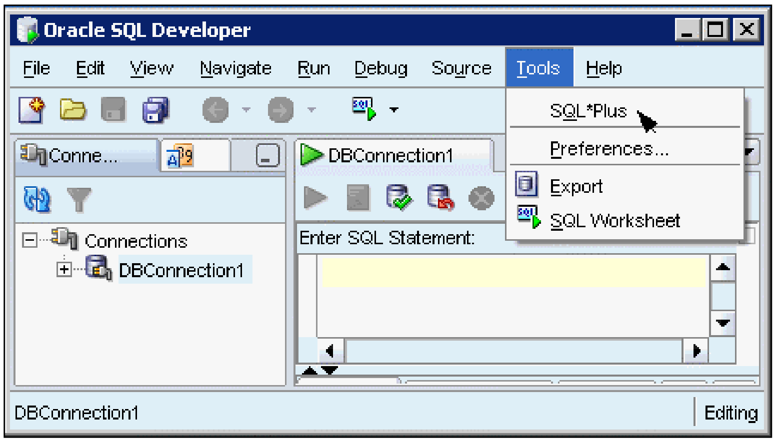
To insert a snippet into your code in a SQL Worksheet or in a PL/SQL function or procedure, drag the snippet from the Snippets window and drop it into the desired place in your code. Then you can edit the syntax so that the SQL function is valid in the current context. To see a brief description of a SQL function in a tool tip, hold the pointer over the function name.

The example in the slide shows that `CONCAT(char1, char2)` is dragged from the Character Functions group in the Snippets window. Then the `CONCAT` function syntax is edited and rest of the statement is added as follows:

```
SELECT CONCAT(first_name, last_name)
FROM employees;
```

Using SQL*Plus

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



Using SQL*Plus

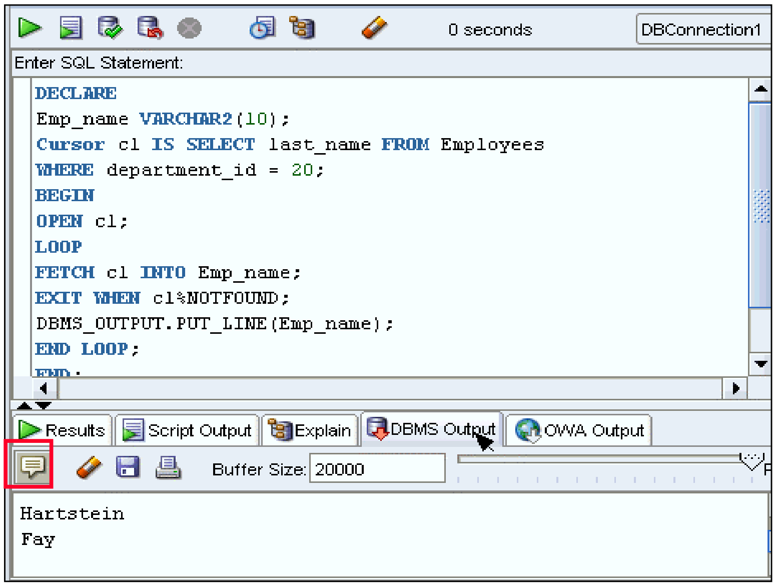
The SQL Worksheet supports some SQL*Plus statements. SQL*Plus statements must be interpreted by the SQL Worksheet before being passed to the database; any SQL*Plus statements that are not supported by the SQL Worksheet are ignored and not passed to the database.

To display the SQL*Plus command window, select **SQL*Plus** from the Tools menu.

To use this feature, the system on which you are using SQL Developer must have an Oracle Home directory or folder, with a SQL*Plus executable under that location. If the location of the SQL*Plus executable is not already stored in your SQL Developer preferences, you are asked to specify its location.

Creating an Anonymous Block

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



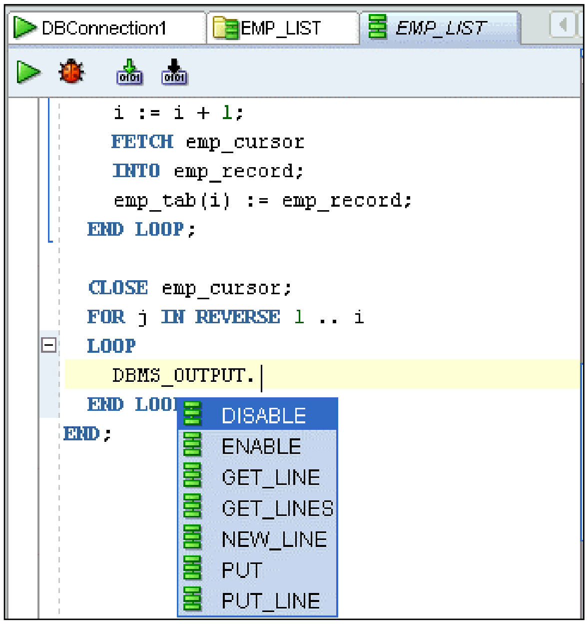
Creating an Anonymous Block

You can create an anonymous block and display the output of `DBMS_OUTPUT` package statements. To create an anonymous block and view the results, perform the following steps:

1. Enter the PL/SQL code in the Enter SQL Statement box.
2. Click the **DBMS Output** pane. Then click the **Enable DBMS Output** icon to set the server output ON.
3. Click the **Execute Statement** icon above the Enter SQL Statement box. Then click the **DBMS Output** pane to see the results.

Editing the PL/SQL Code

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



Editing the PL/SQL Code

You may want to make changes to your PL/SQL code. SQL Developer includes a full-featured editor for PL/SQL program units. It includes customizable PL/SQL syntax highlighting in addition to common editor functions such as:

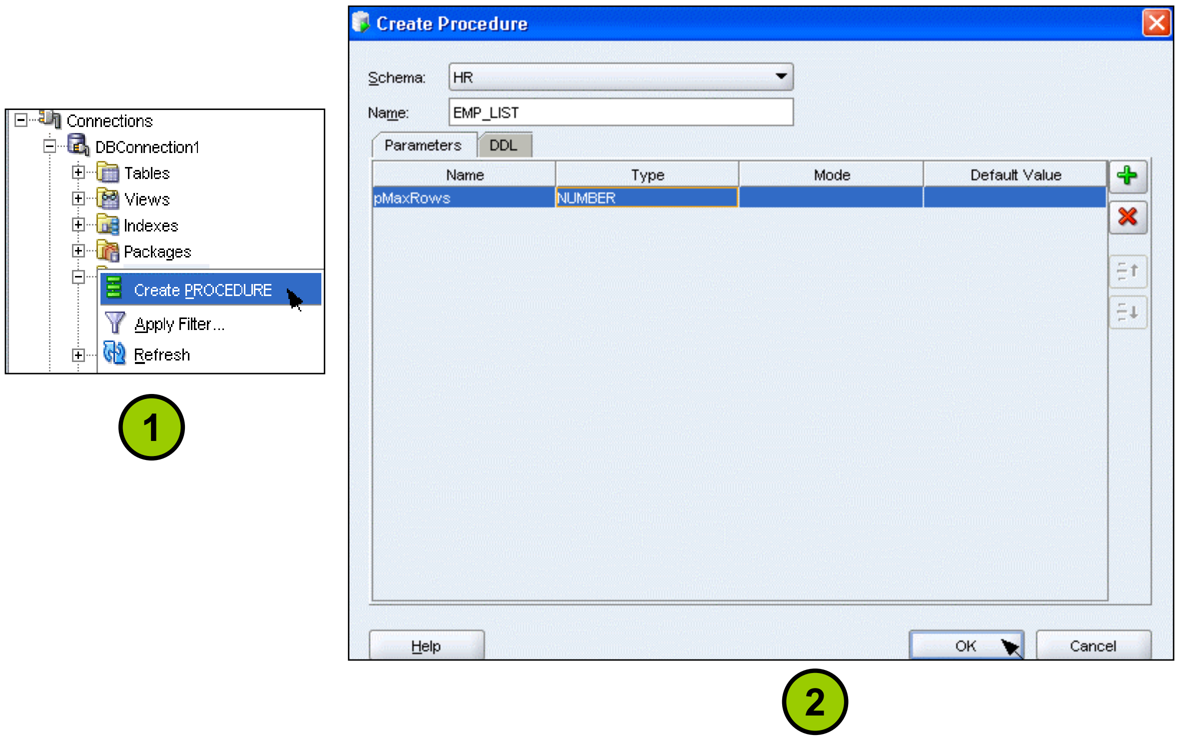
- Bookmarks
- Code Completion
- Code Folding
- Search and Replace

To edit the PL/SQL code, click the object name in the Connections navigator, and then click the **Edit** icon. Optionally, double-click the object name to invoke the object definition page with its tabs and the Edit page. You can update only if you are in the Edit tab.

The Code Insight feature is shown on the slide. For example, if you type `DBMS_OUTPUT.` and then press [Ctrl] + [Space], you can select from a list of members of that package. Note that by default, Code Insight is invoked automatically if you pause after typing a period (.) for more than one second.

When using the Code Editor to edit PL/SQL code, you can “Compile” or “Compile for Debug.”

Creating a PL/SQL Procedure



Creating a PL/SQL Procedure

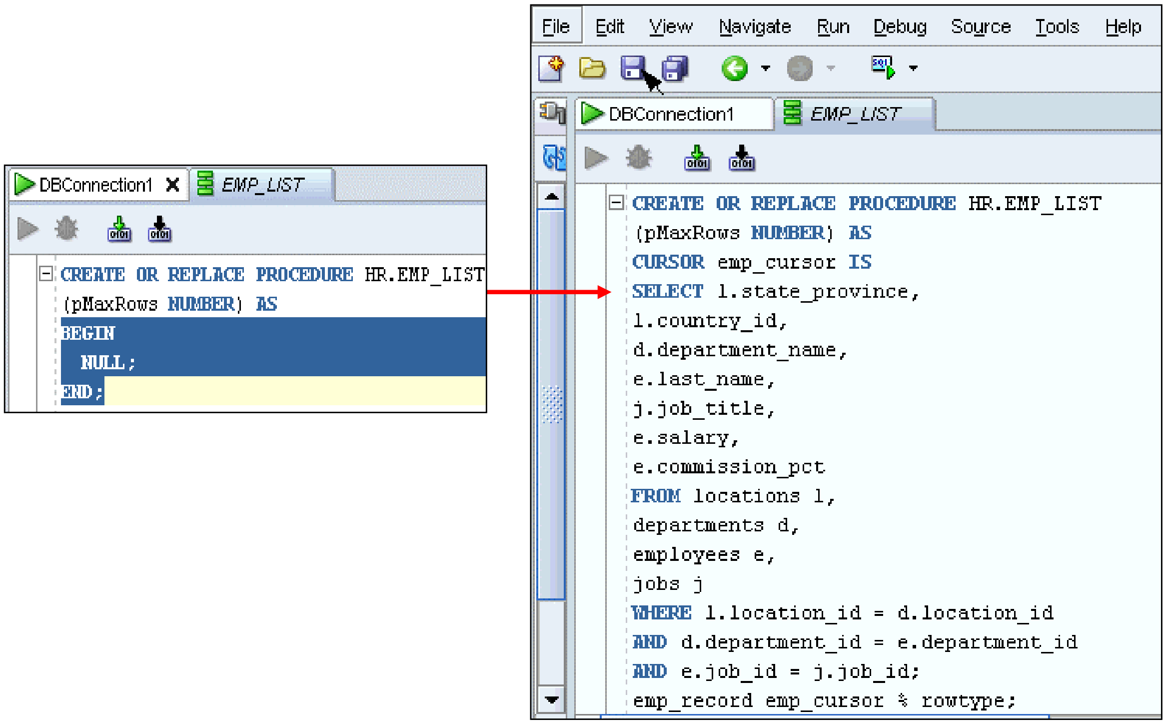
Using SQL Developer, you can create PL/SQL functions, procedures, and packages. To create a Pl/SQL procedure, perform the following steps:

1. Right-click the Procedures node in the Connections Navigator to invoke the context menu, and select **Create Procedure**.
2. In the Create Procedure dialog box, specify the procedure information and click **OK**.

Note: Ensure that you press Enter before you click OK.

In the example in the slide, the `EMP_LIST` procedure is created. The default values for parameter name and parameter type are replaced with `pMaxRows` and `NUMBER` respectively.

Compiling a PL/SQL Procedure



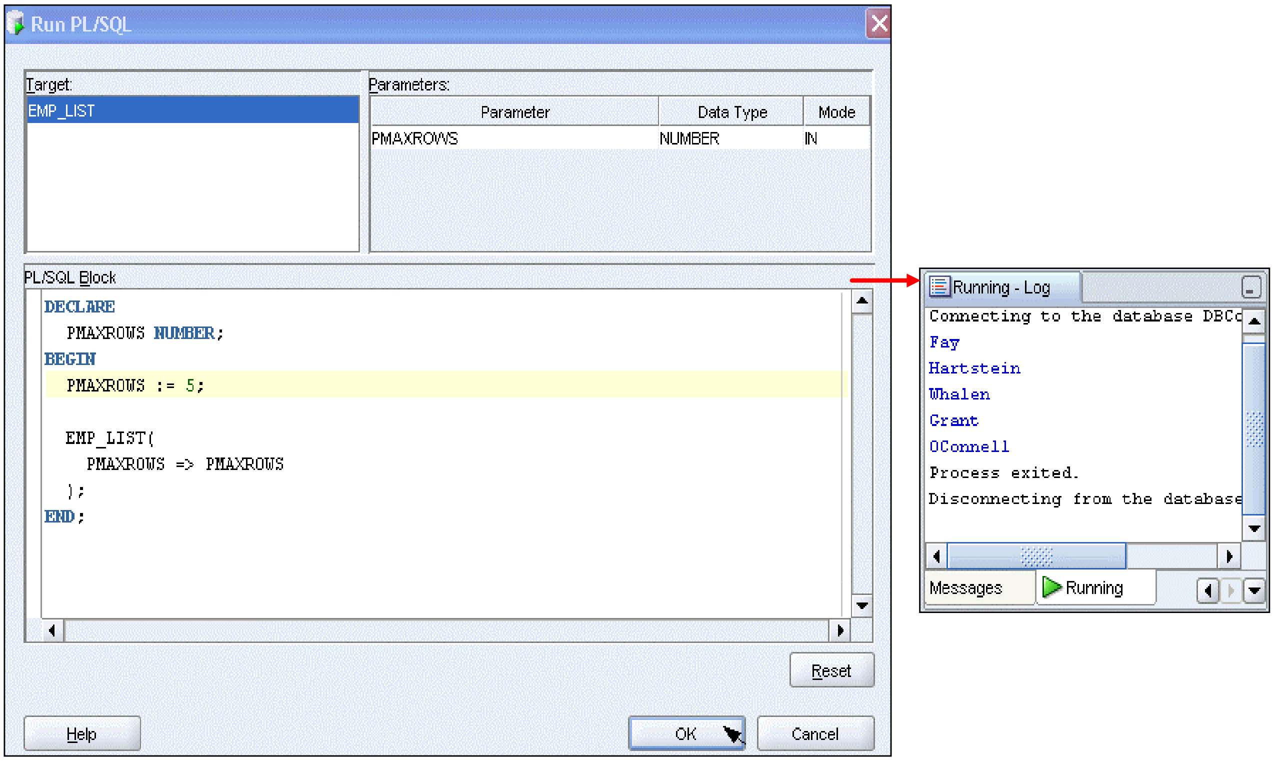
Compiling a PL/SQL Procedure

Once you specify the parameter information in the Create Procedure dialog box and click OK, you see the procedure tab added in the right window. You can then replace the Anonymous block with your PL/SQL code.

To compile the PL/SQL subprogram, click the Save button in the toolbar. If you expand Procedures in the Connections Navigator, you can see that the procedure node is added.

When an invalid PL/SQL subprogram is detected by SQL Developer, the status is indicated with a red X over the icon for the subprogram in the Connections Navigator. Compilation errors are shown in the log window. You can navigate to the line reported in the error by simply double-clicking on the error. SQL Developer also displays errors and hints in the right hand gutter. If you hover each of the red bars in the gutter, the error message displays. For example, if the error messages indicate that there is a formatting error, modify the code accordingly and click the Compile icon.

Running a PL/SQL Procedure



Running a PL/SQL Procedure

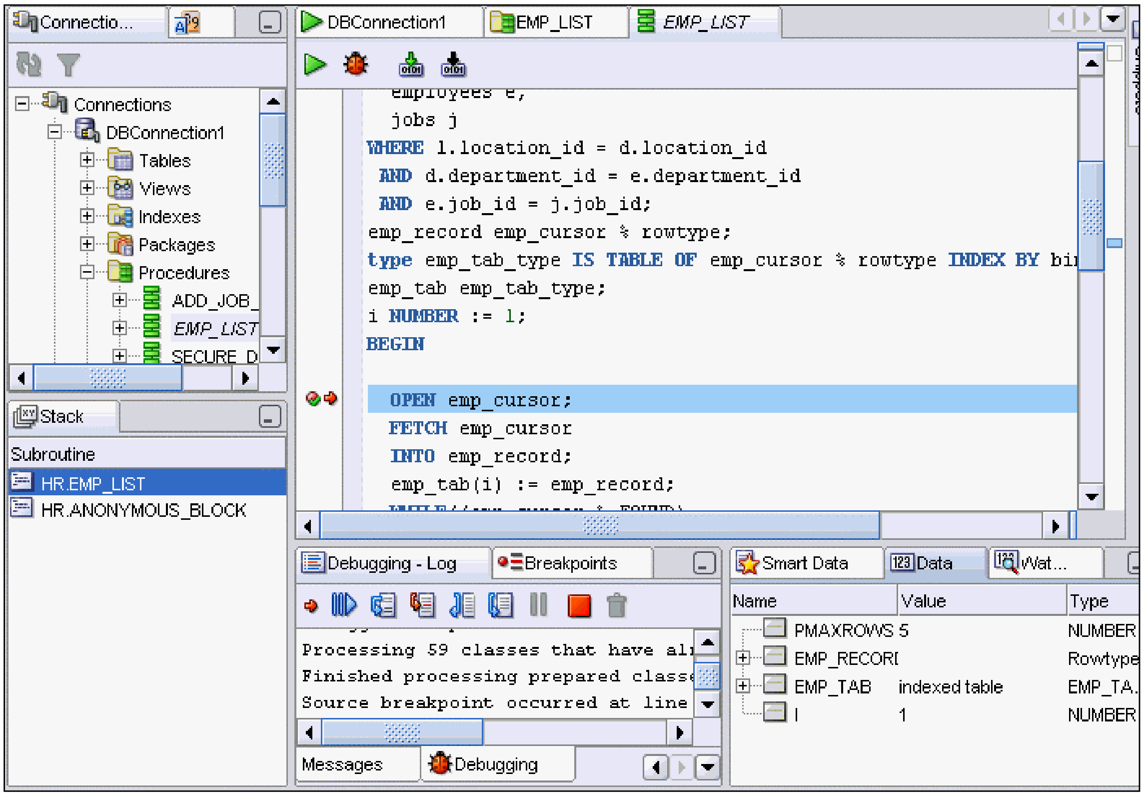
Once you have created and compiled a PL/SQL procedure, you can run it using SQL Developer. To run a PL/SQL procedure, right-click the procedure name in the left navigator and select Run. Optionally, you can use the Run button in the right window. This invokes the Run PL/SQL dialog box. The Run PL/SQL dialog box allows you to select the target procedure or function to run and displays a list of parameters for the selected target.

You can use the PL/SQL block area to populate parameters to be passed to the program unit and to handle complex return types. Once you make the necessary changes in the Run PL/SQL dialog box, click **OK**. You see the expected results in the Running-Log window.

In the example in the slide, `PMAXROWS := NULL;` is changed to `PMAXROWS := 5;`

The results of the five rows returned are displayed in the Running-Log window.

Debugging PL/SQL



Debugging PL/SQL

You may want to debug a PL/SQL function, procedure or package. SQL Developer provides full support for PL/SQL debugging. To debug a function or procedure, perform the following steps:

1. Click the object name in the Connections navigator
2. Right-click the object and select **Compile for debug**.
3. Click the **Edit** icon. Then click the **Debug** icon above its source listing.

If the toggle numbers before each line of code is not yet displayed, right-click in the Code Editor margin and select Toggle Line Numbers.

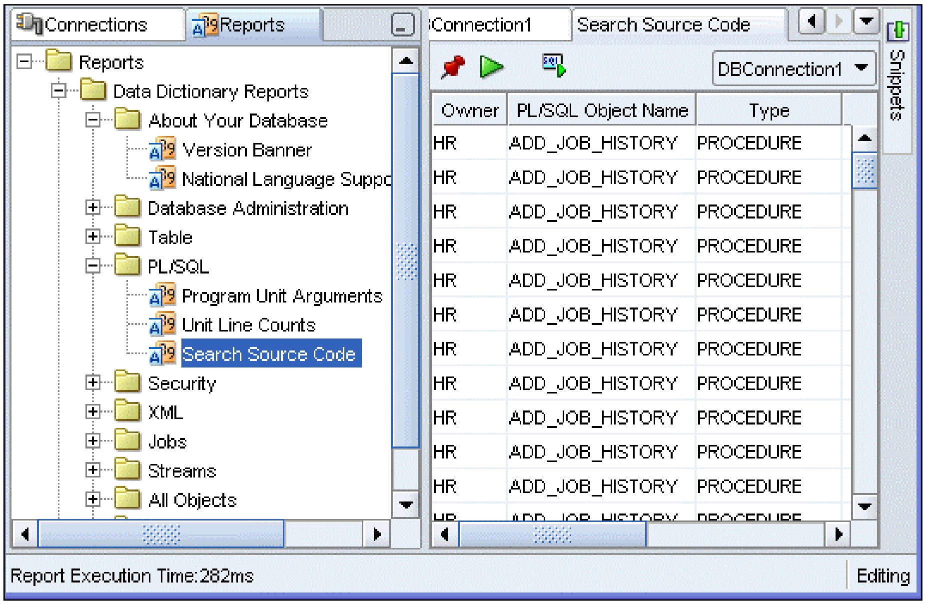
The PL/SQL debugger supplies many commands to control program execution including Step Into, Step Over, Step Out, Run to Cursor, and so on. While the debugger is paused, you can examine and modify the values of variables from the Smart Data, Watches or Inspector windows.

The Breakpoints window lists the defined breakpoints. You can use this window to add new breakpoints, or customize the behavior of existing breakpoints.

Note: For PL/SQL debugging, you need the `debug any procedure` and `debug connect session` privileges.

Database Reporting

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



Database Reporting

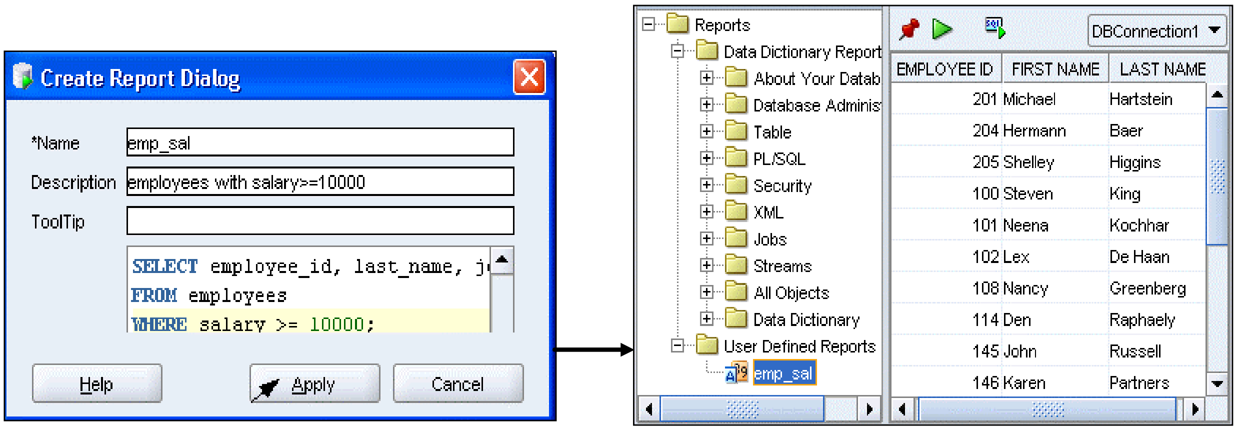
SQL Developer provides many reports about the database and its objects. These reports can be grouped into the following categories:

- About Your Database reports
- Database Administration reports
- Table reports
- PL/SQL reports
- Security reports
- XML reports
- Jobs reports
- Streams reports
- All Objects reports
- Data Dictionary reports
- User Defined reports

To display reports, click the Reports tab on the left side of the window. Individual reports are displayed in tabbed panes on the right side of the window; and for each report, you can select (in a drop-down control) the database connection for which to display the report. For reports about objects, the objects shown are only those visible to the database user associated with the selected database connection, and the rows are usually ordered by Owner.

Creating a User Defined Report

Create and save user-defined reports for repeated use:



Creating a User Defined Report

User Defined reports are any reports that are created by SQL Developer users. To create a user-defined report, perform the following steps:

1. Right-click the **User Defined Reports** node under Reports, and select **Add Report**.
2. In the Create Report Dialog box, specify the report name and the SQL query to retrieve information for the report. Then click **Apply**.

In the example in the slide, the report name is specified as `emp_sal`. An optional description is provided indicating that the report contains details of employees with salary >= 10000. The complete SQL statement for retrieving the information to be displayed in the user-defined report is specified in the SQL box. You can also include an optional tool tip to be displayed when the mouse pointer stays briefly over the report name in the Reports navigator display.

You can organize user-defined reports in folders, and you can create a hierarchy of folders and subfolders. To create a folder for user-defined reports, right-click the User Defined node or any folder name under that node and select Add Folder.

Information about user-defined reports, including any folders for these reports, is stored in a file named `UserReports.xml` under the directory for user-specific information.

Summary

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

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

Summary

SQL Developer is a free graphical tool to simplify database development tasks. Using SQL Developer, you can browse, create, and edit database objects. You can use the SQL Worksheet to run SQL statements and scripts. Using SQL Developer, you can edit and debug PL/SQL.

SQL Developer enables you to create and save your own special set of reports for repeated use.

Index