Introduction to Oracle9*i*: PL/SQL

Student Guide . Volume 1

40054GC10
Production 1.0
June 2001
D32945

ORACLE®

Authors

Nagavalli Pataballa
Priya Nathan

Technical Contributors and Reviewers

Anna Atkinson
Bryan Roberts
Caroline Pereda
Cesljas Zarco
Chaya Rao
Coley William
Daniel Gabel
Dr. Christoph Burandt
Hakan Lindfors
Helen Robertson
John Hoff
Judy Brink
Lachlan Williams
Laszlo Czinkoczki
Laura Pezzini
Linda Boldt
Marco Verbeek
Natarajan Senthil
Priya Vennapusa
Robert Squires
Roger Abuzalaf
Ruediger Steffan
Sarah Jones
Stefan Lindblad
Sue Onraet
Susan Dee

Publisher

Sandya Krishna

Contents

17 More Trigger Concepts

Preface

Profile

Before You Begin This Course

Before you begin this course, you should have thorough knowledge of SQL, *i*SQL*Plus, and working experience developing applications. Required prerequisites are *Introduction to Oracle9i: SQL*, or *Introduction to Oracle9i for Experienced SQL Users.*

How This Course Is Organized

Introduction to Oracle9i: PL/SQL 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

Oracle Publications

Title	Part Number
Oracle9i Application Developer's Guide-Fundamentals	*A86797-01*
Oracle9i Application Developer's Guide-Large Objects	*A86800-01*
Oracle9i Supplied PL/SQL Packages Reference	*A86815-01*
PL/SQL User's Guide and Reference, Release 8.1.6	*A86811-01*

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, table names, PL/SQL objects, schemas	Use the SELECT command to view information stored in the LAST_NAME column of the EMPLOYEES table.
Lowercase,	Filenames, syntax variables, usernames, passwords	where: *role* is the name of the role italic to be created.
Initial cap	Trigger and button names	Assign a When-Validate-Item trigger to the ORD block. Choose Cancel.
Italic	Books, names of courses and manuals, and emphasized words or phrases	For more information on the subject, see *Oracle8 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

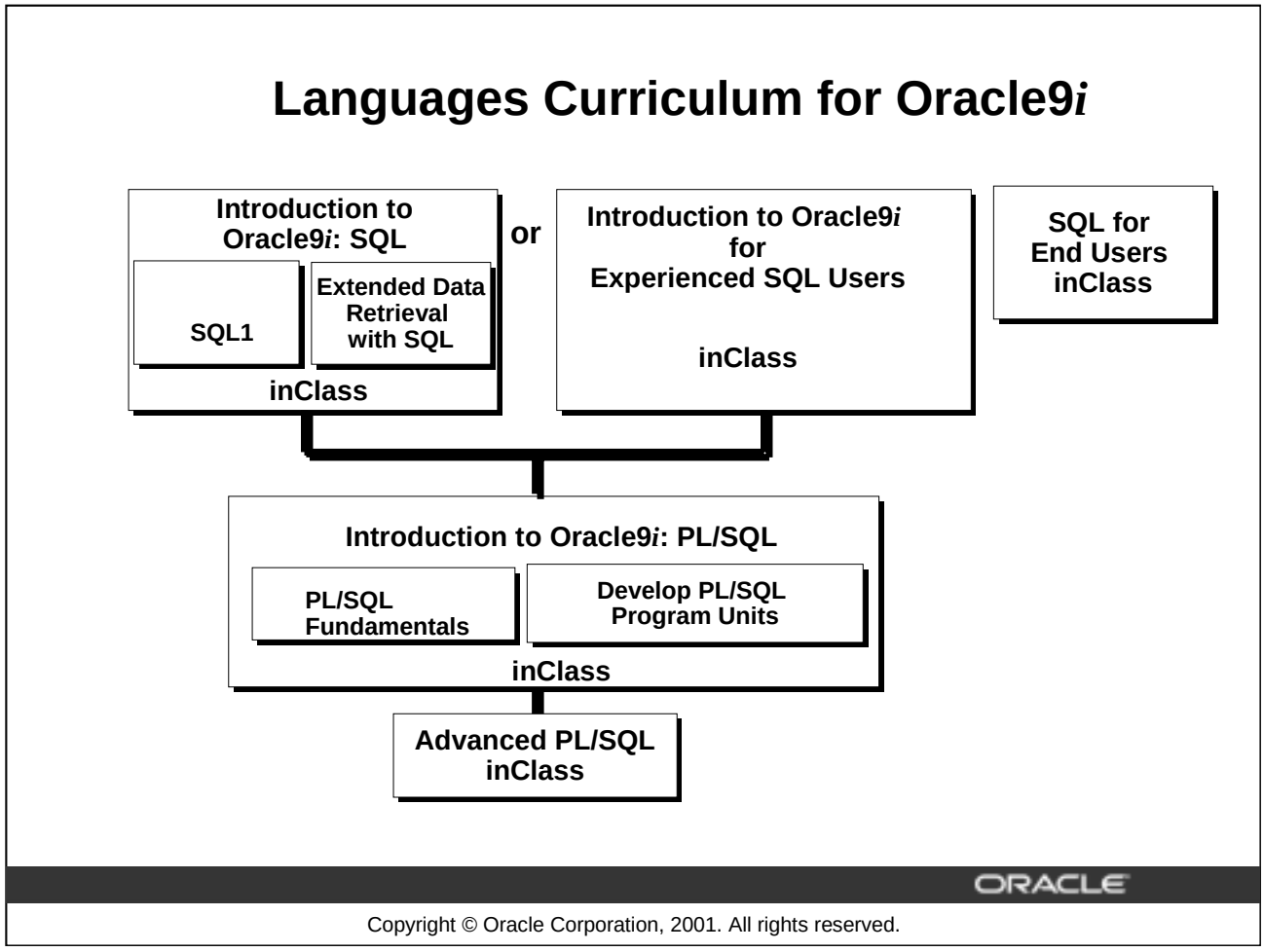
Convention	Object or Term	Example
Uppercase	Commands, functions	`SQL> SELECT userid` `  2  FROM emp;`
Lowercase, italic	Syntax variables	`SQL> CREATE ROLE` *`role`*`;`

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'))`
		`. . .`
		`SQL> SELECT last_name`
		`  2  FROM emp;`

Bold	Text that must be entered by a user	`SQLDBA> DROP USER scott` `  2> IDENTIFIED BY tiger;`

Curriculum Map



Integrated Languages Curriculum

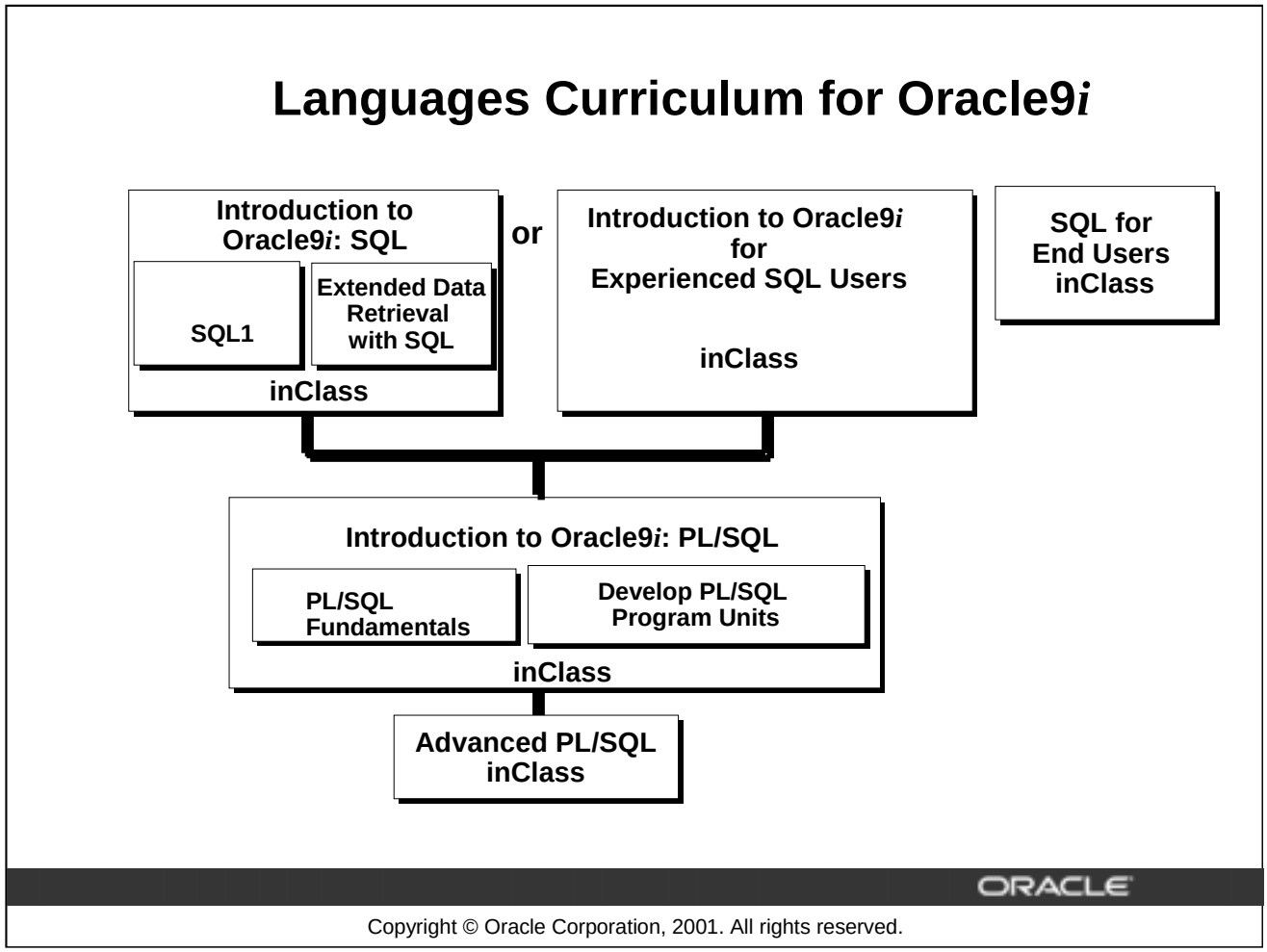
Introduction to Oracle9i: SQL consists of two modules, *SQL1* and *Extended Data Retrieval with SQL*. *SQL1* covers creating database structures and storing, retrieving, and manipulating data in a relational database. *Extended Data Retrieval with SQL* covers advanced SELECT statements, Oracle SQL and *i*SQL*Plus Reporting.

For people who have worked with other relational databases and have knowledge of SQL, another course, called *Introduction to Oracle9i for Experienced SQL Users* is offered. This course covers the SQL statements that are not part of ANSI SQL but are specific to Oracle.

Introduction to Oracle9i: PL/SQL consists of two modules, *PL/SQL Fundamentals* and *Develop PL/SQL Program Units*. *PL/SQL Fundamentals* covers PL/SQL basics including the PL/SQL language structure, flow of execution and interface with SQL. *Develop PL/SQL Program Units* covers how to create stored procedures, functions, packages, and triggers as well as maintain and debug program code.

SQL for End Users is directed towards individuals with little programming background and covers basic SQL statements. This course is for end users who need to know some basic SQL programming.

Advanced PL/SQL is appropriate for individuals who have experience in PL/SQL programming and covers coding efficiency topics, object-oriented programming, working with external code, and the advanced features of the Oracle supplied packages.



Integrated Languages Curriculum

The slide lists various modules and courses that are available in the languages curriculum. The following table lists the modules and courses with their equivalent TBTs.

Course or Module	Equivalent TBT
SQL1	*Oracle SQL: Basic SELECT Statements* *Oracle SQL: Data Retrieval Techniques* *Oracle SQL: DML and DDL*
Extended Data Retrieval with SQL	*Oracle SQL and SQL*Plus: Advanced SELECT Statements* *Oracle SQL and SQL*Plus: SQL*Plus and Reporting*
Introduction to Oracle9i for Experienced SQL Users	*Oracle SQL Specifics: Retrieving and Formatting Data* *Oracle SQL Specifics: Creating and Managing Database Objects*
PL/SQL Fundamentals	*PL/SQL: Basics*
Develop PL/SQL Program Units	*PL/SQL: Procedures, Functions, and Packages* *PL/SQL: Database Programming*
SQL for End Users	*SQL for End Users: Part 1* *SQL for End Users: Part 2*
Advanced PL/SQL	*Advanced PL/SQL: Implementation and Advanced Features* *Advanced PL/SQL: Design Considerations and Object Types*

Overview of PL/SQL

ORACLE

Course Objectives

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

- **Describe the purpose of PL/SQL**
- **Describe the use of PL/SQL for the developer as well as the DBA**
- **Explain the benefits of PL/SQL**
- **Create, execute, and maintain procedures, functions, packages, and database triggers**
- **Manage PL/SQL subprograms and triggers**
- **Describe Oracle supplied packages**
- **Manipulate large objects (LOBs)**

ORACLE

Lesson Aim

In this course, you are introduced to the features and benefits of PL/SQL. You learn how to access the database using PL/SQL.

You can develop modularized applications with database procedures using database objects, such as the following:

- Procedures and functions
- Packages
- Database triggers

Modular applications improve:

- Functionality
- Security
- Overall performance

About PL/SQL

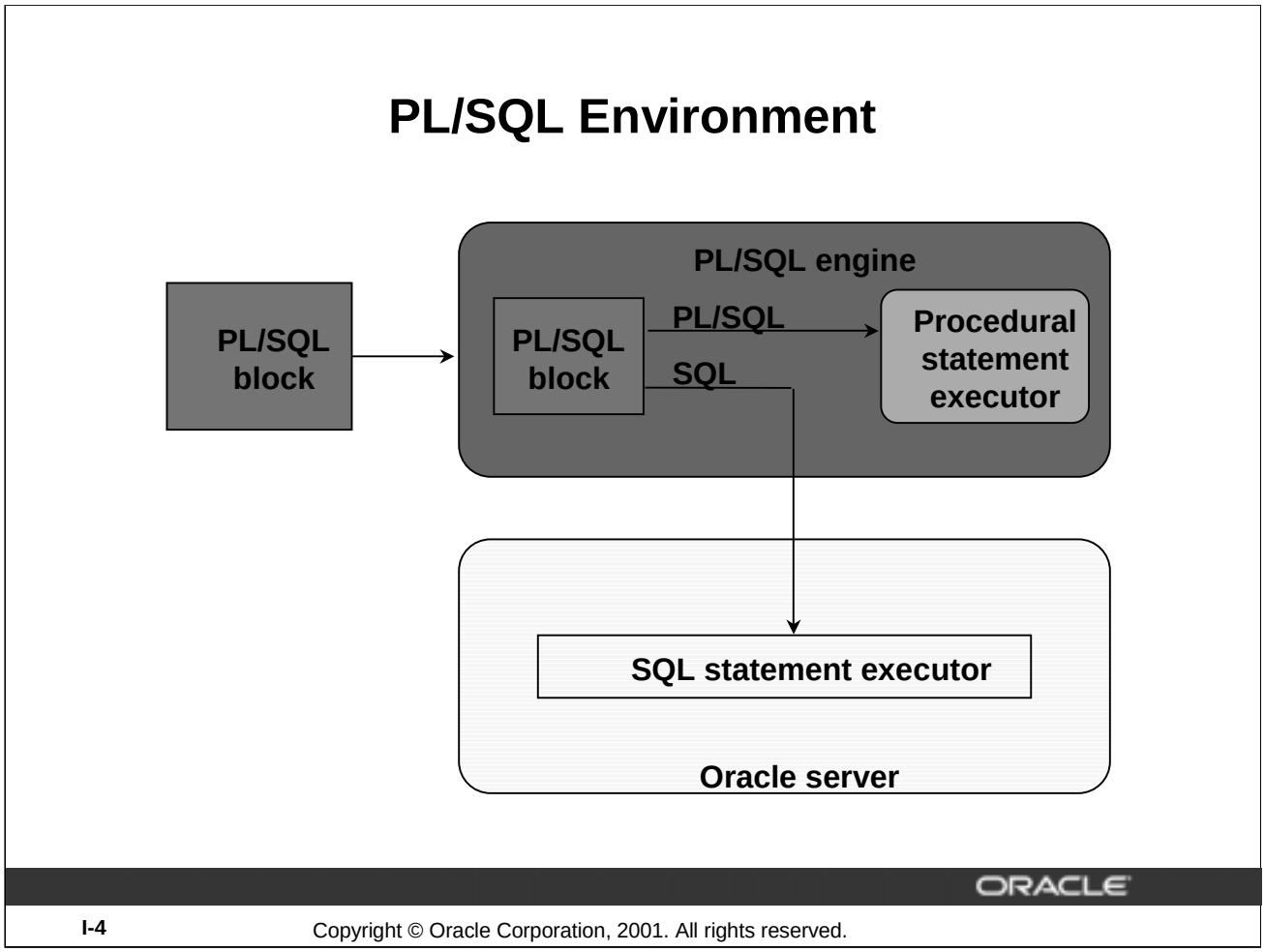
- **PL/SQL is the procedural extension to SQL with design features of programming languages.**
- **Data manipulation and query statements of SQL are included within procedural units of code.**

ORACLE

About PL/SQL

Procedural Language/SQL (PL/SQL) is Oracle Corporation's procedural language extension to SQL, the standard data access language for relational databases. PL/SQL offers modern software engineering features such as data encapsulation, exception handling, information hiding, object orientation, and brings state-of-the-art programming to the Oracle Server and toolset.

PL/SQL incorporates many of the advanced features of programming languages that were designed during the 1970s and 1980s. It allows the data manipulation and query statements of SQL to be included in block-structured and procedural units of code, making PL/SQL a powerful transaction processing language. With PL/SQL, you can use SQL statements to finesse Oracle data, and PL/SQL control statements to process the data.



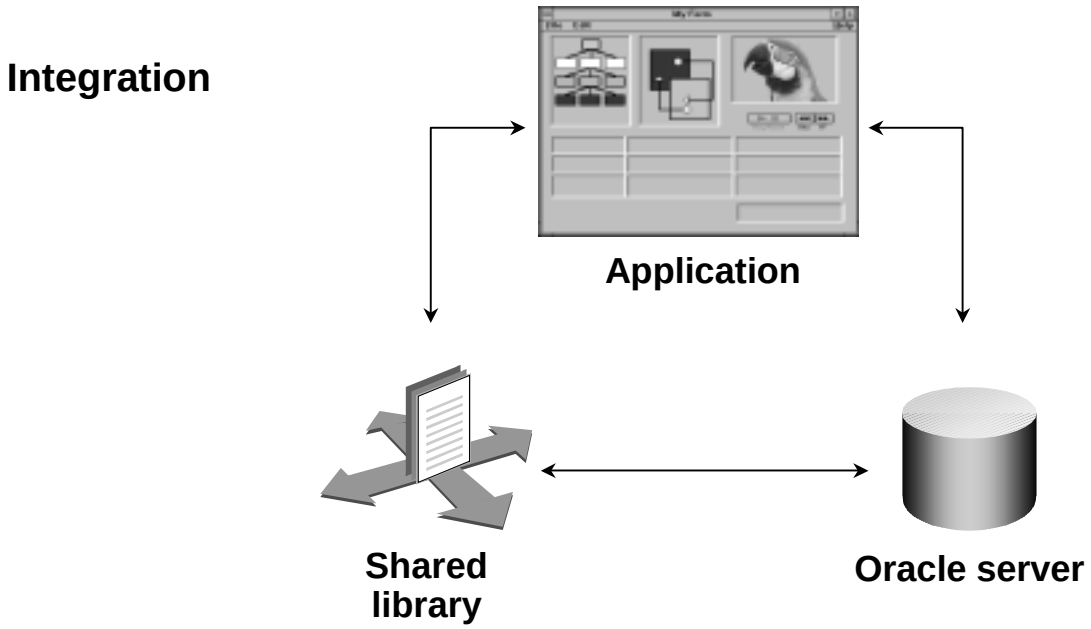
PL/SQL Environment

PL/SQL is not an Oracle product in its own right; it is a technology used by the Oracle server and by certain Oracle tools. Blocks of PL/SQL are passed to and processed by a PL/SQL engine, which may reside within the tool or within the Oracle server. The engine that is used depends on where the PL/SQL block is being invoked from.

When you submit PL/SQL blocks from a Pro*C or a Pro*Cobol program, user-exit, *i*SQL*Plus, or Server Manager, the PL/SQL engine in the Oracle Server processes them. It separates the SQL statements and sends them individually to the SQL statements executor.

A single transfer is required to send the block from the application to the Oracle Server, thus improving performance, especially in a client-server network. PL/SQL code can also be stored in the Oracle Server as subprograms that can be referenced by any number of applications connected to the database.

Benefits of PL/SQL



ORACLE

Benefits of PL/SQL

Integration:

PL/SQL plays a central role in both the Oracle server (through stored procedures, stored functions, database triggers, and packages) and Oracle development tools (through Oracle Developer component triggers).

Oracle Forms Developer, Oracle Reports Developer, and Oracle Graphics Developer applications make use of shared libraries that hold code (procedures and functions) and can be accessed locally or remotely.

SQL data types can also be used in PL/SQL. Combined with the direct access that SQL provides, these shared data types integrate PL/SQL with the Oracle server data dictionary. PL/SQL bridges the gap between convenient access to database technology and the need for procedural programming capabilities.

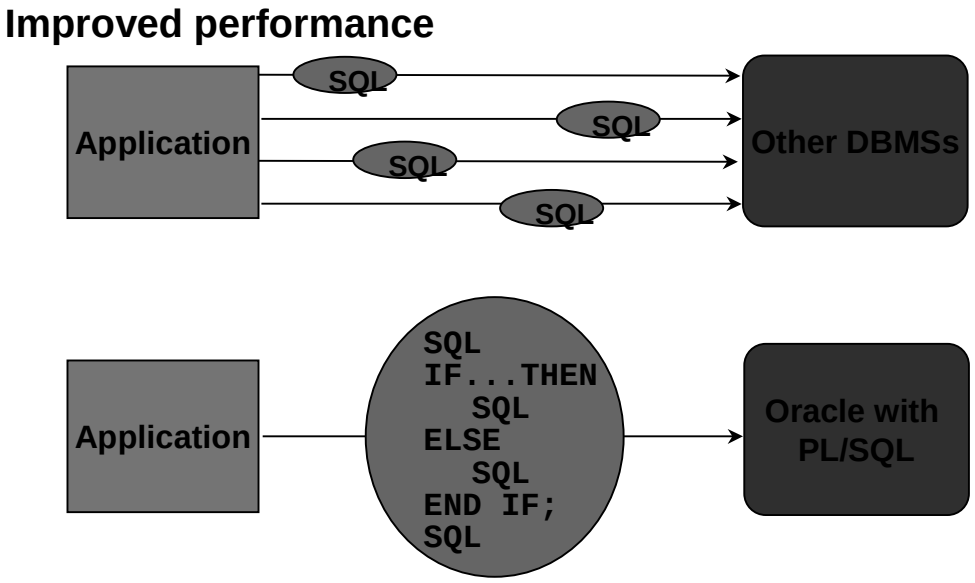
PL/SQL in Oracle Tools:

Many Oracle tools, including Oracle Developer, have their own PL/SQL engine, which is independent of the engine present in the Oracle Server.

The engine filters out SQL statements and sends them individually to the SQL statement executor in the Oracle server. It processes the remaining procedural statements in the procedural statement executor, which is in the PL/SQL engine.

The procedural statement executor processes data that is local to the application (that is, data already inside the client environment, rather than in the database). This reduces the work that is sent to the Oracle server and the number of memory cursors that are required.

Benefits of PL/SQL



ORACLE

Benefits of PL/SQL (continued)

Improved Performance:

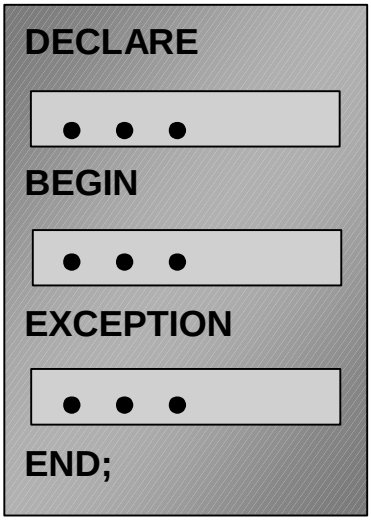
PL/SQL can improve the performance of an application. The benefits differ depending on the execution environment.

- PL/SQL can be used to group SQL statements together within a single block and to send the entire block to the server in a single call, thereby reducing networking traffic. Without PL/SQL, the SQL statements are sent to the Oracle server one at a time. Each SQL statement results in another call to the Oracle server and higher performance overhead. In a networked environment, the overhead can become significant. As the slide illustrates, if the application is SQL intensive, you can use PL/SQL blocks and subprograms to group SQL statements before sending them to the Oracle server for execution.
- PL/SQL can also operate with Oracle Server application development tools such as Oracle Forms and Oracle Reports. By adding procedural processing power to these tools, PL/SQL enhances performance.

Note: Procedures and functions that are declared as part of a Developer application are distinct from those stored in the database, although their general structure is the same. Stored subprograms are database objects and are stored in the data dictionary. They can be accessed by any number of applications, including Developer applications.

Benefits of PL/SQL

Modularize program development



ORACLE

Benefits of PL/SQL (continued)

You can take advantage of the procedural capabilities of PL/SQL, which are not available in SQL.

PL/SQL Block Structure:

Every unit of PL/SQL comprises one or more blocks. These blocks can be entirely separate or nested one within another. The basic units (procedures, functions, and anonymous blocks) that make up a PL/SQL program are logical blocks, which can contain any number of nested subblocks. Therefore, one block can represent a small part of another block, which in turn can be part of the whole unit of code.

Modularized Program Development:

- Group logically related statements within blocks.
- Nest subblocks inside larger blocks to build powerful programs.
- Break down a complex problem into a set of manageable, well-defined, logical modules and implement the modules with blocks.
- Place reusable PL/SQL code in libraries to be shared between Oracle Forms and Oracle Reports applications or store it in an Oracle server to make it accessible to any application that can interact with an Oracle database.

Benefits of PL/SQL

- **PL/SQL is portable.**
- **You can declare variables.**

ORACLE

Benefits of PL/SQL (continued)

Portability:

- Because PL/SQL is native to the Oracle server, you can move programs to any host environment (operating system or platform) that supports the Oracle server and PL/SQL. In other words, PL/SQL programs can run anywhere the Oracle server can run; you do not need to tailor them to each new environment.
- You can also move code between the Oracle server and your application. You can write portable program packages and create libraries that can be reused in different environments.

Identifiers:

In PL/SQL you can use identifiers to do the following:

- Declare variables, cursors, constants, and exceptions and then use them in SQL and procedural statements
- Declare variables belonging to scalar, reference, composite, and large object (LOB) data types
- Declare variables dynamically based on the data structure of tables and columns in the database

Benefits of PL/SQL

- **You can program with procedural language control structures.**
- **PL/SQL can handle errors.**

ORACLE

Benefits of PL/SQL (continued)

Procedural Language Control Structures:

Procedural Language Control Structures allow you to do the following:

- Execute a sequence of statements conditionally
- Execute a sequence of statements iteratively in a loop
- Process individually the rows returned by a multiple-row query with an explicit cursor

Errors:

The Error handling functionality in PL/SQL allows you to do the following:

- Process Oracle server errors with exception-handling routines
- Declare user-defined error conditions and process them with exception-handling routines

Benefits of Subprograms

- **Easy maintenance**
- **Improved data security and integrity**
- **Improved performance**
- **Improved code clarity**

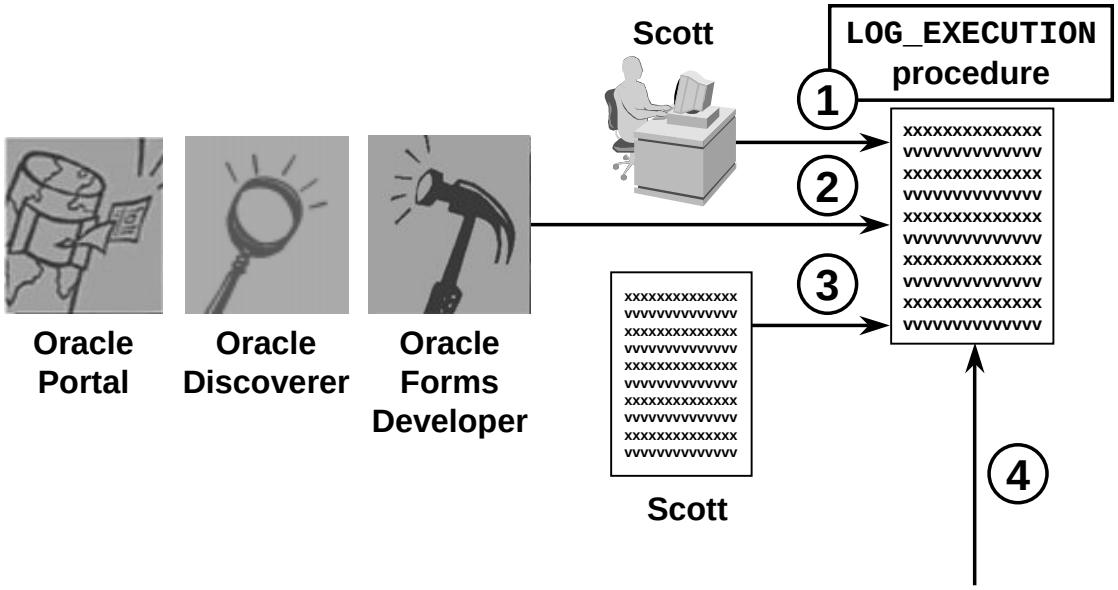
ORACLE

Benefits of Subprograms

Stored procedures and functions have many benefits in addition to modularizing application development:

- Easy maintenance that enables you to modify:
  - Routines online without interfering with other users
  - One routine to affect multiple applications
  - One routine to eliminate duplicate testing
- Improved data security and integrity by doing the following:
  - Control indirect access to database objects from nonprivileged users with security privileges
  - Ensure that related actions are performed together, or not at all, by funneling activity for related tables through a single path
- Improved performance that allows you to do the following:
  - Avoid reparsing for multiple users by exploiting the shared SQL area
  - Avoid PL/SQL parsing at run time by parsing at compilation time
  - Reduce the number of calls to the database and decrease network traffic by bundling commands
- Improved code clarity: Using appropriate identifier names to describe the action of the routines reduces the need for comments and enhances the clarity of the code.

Invoking Stored Procedures and Functions



How to Invoke Stored Procedures and Functions

You can invoke a previously created procedure or function from a variety of environments such as *i*SQL*Plus, Oracle Forms Developer, Oracle Discoverer, Oracle Portal, another stored procedure, and many other Oracle tools and precompiler applications. The table below describes how you can invoke a previously created procedure, `log_execution`, from a variety of environments.

*i*SQL*Plus	`EXECUTE log_execution`
Oracle development tools such as Oracle Forms Developer	`log_execution;`

Another procedure	`CREATE OR REPLACE PROCEDURE leave_emp`
	`(v_id IN employees.employee_id%TYPE)`
	`IS`
	`BEGIN`
	`DELETE FROM employees`
	`WHERE employee_id = v_id;`
	`log_execution;`
	`END`
	`leave_emp;`

Other environments such as Pro*C	

Summary

- PL/SQL is an extension to SQL.
- Blocks of PL/SQL code are passed to and processed by a PL/SQL engine.
- Benefits of PL/SQL:
  - Integration
  - Improved performance
  - Portability
  - Modularity of program development
- Subprograms are named PL/SQL blocks, declared as either procedures or functions.
- You can invoke subprograms from different environments.

ORACLE

Summary

PL/SQL is a language that has programming features that serve as an extension to SQL. It provides you with the ability to control the flow of constructs, and declare and use variables. PL/SQL applications can run on any platform or operating system on which Oracle runs.

Named PL/SQL blocks are also known as subprograms or program units. Procedures, functions, packages, and triggers are different PL/SQL constructs. You can invoke subprograms from different environments.

1 Declaring Variables

ORACLE

Objectives

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

- **Recognize the basic PL/SQL block and its sections**
- **Describe the significance of variables in PL/SQL**
- **Declare PL/SQL variables**
- **Execute a PL/SQL block**

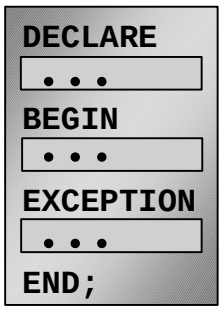
ORACLE

Lesson Aim

This lesson presents the basic rules and structure for writing and executing PL/SQL blocks of code. It also shows you how to declare variables and assign data types to them.

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
```

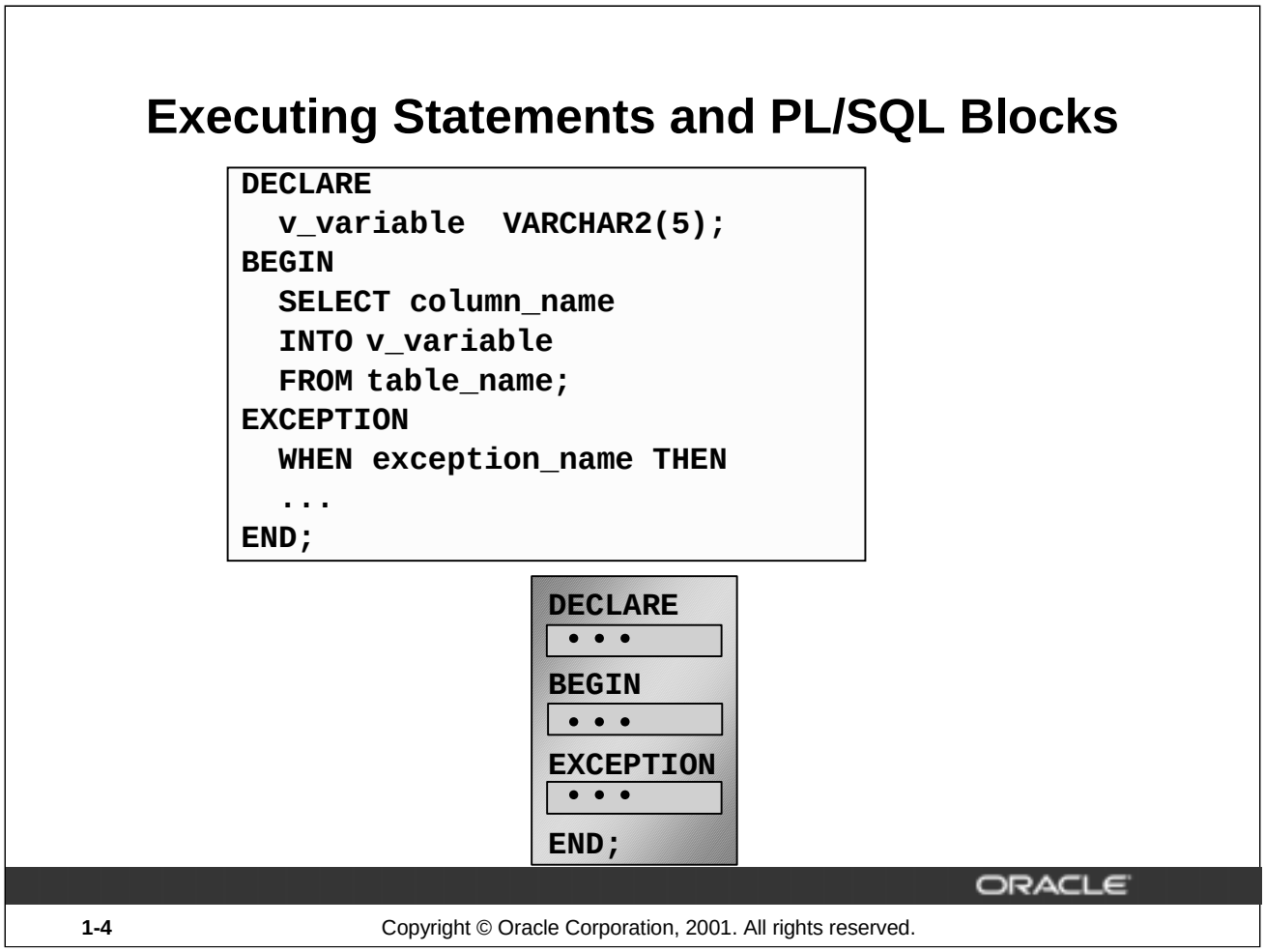


ORACLE

PL/SQL Block Structure

PL/SQL is a block-structured language, meaning that programs can be divided into logical blocks. A PL/SQL block consists of up to three sections: declarative (optional), executable (required), and exception handling (optional). The following table describes the three sections:

Section	Description	Inclusion
Declarative	Contains all variables, constants, cursors, and user-defined exceptions that are referenced in the executable and declarative sections	Optional
Executable	Contains SQL statements to manipulate data in the database and PL/SQL statements to manipulate data in the block	Mandatory
Exception handling	Specifies the actions to perform when errors and abnormal conditions arise in the executable section	Optional



Executing Statements and PL/SQL Blocks

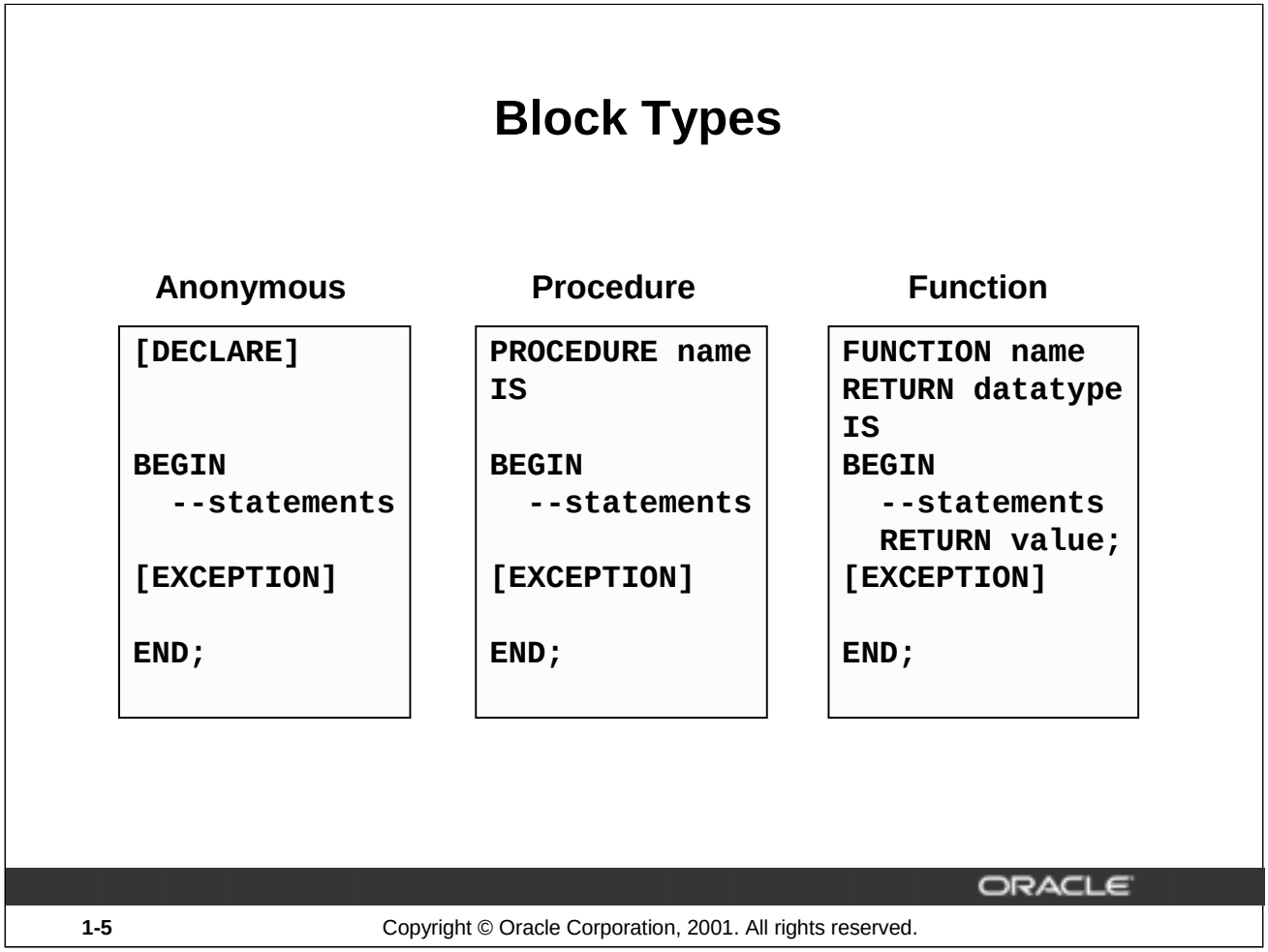
- Place a semicolon (;) at the end of a SQL statement or PL/SQL control statement.
- When the block is executed successfully, without unhandled errors or compile errors, the message output should be as follows:

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

- Section keywords `DECLARE`, `BEGIN`, and `EXCEPTION` are not followed by semicolons.
- `END` and all other PL/SQL statements require a semicolon to terminate the statement.
- You can string statements together on the same line, but this method is not recommended for clarity or editing.

Note: In PL/SQL, an error is called an exception.

With modularity you can break an application down into manageable, well-defined modules. Through successive refinement, you can reduce a complex problem to a set of simple problems that have easy-to-implement solutions. PL/SQL meets this need with program units, which include blocks, subprograms, and packages.



Block Types

A PL/SQL program comprises one or more blocks. These blocks can be entirely separate or nested one within another. The basic units (procedures and functions, also known as subprograms, and anonymous blocks) that make up a PL/SQL program are logical blocks, which can contain any number of nested subblocks. Therefore, one block can represent a small part of another block, which in turn can be part of the whole unit of code.

Anonymous Blocks:

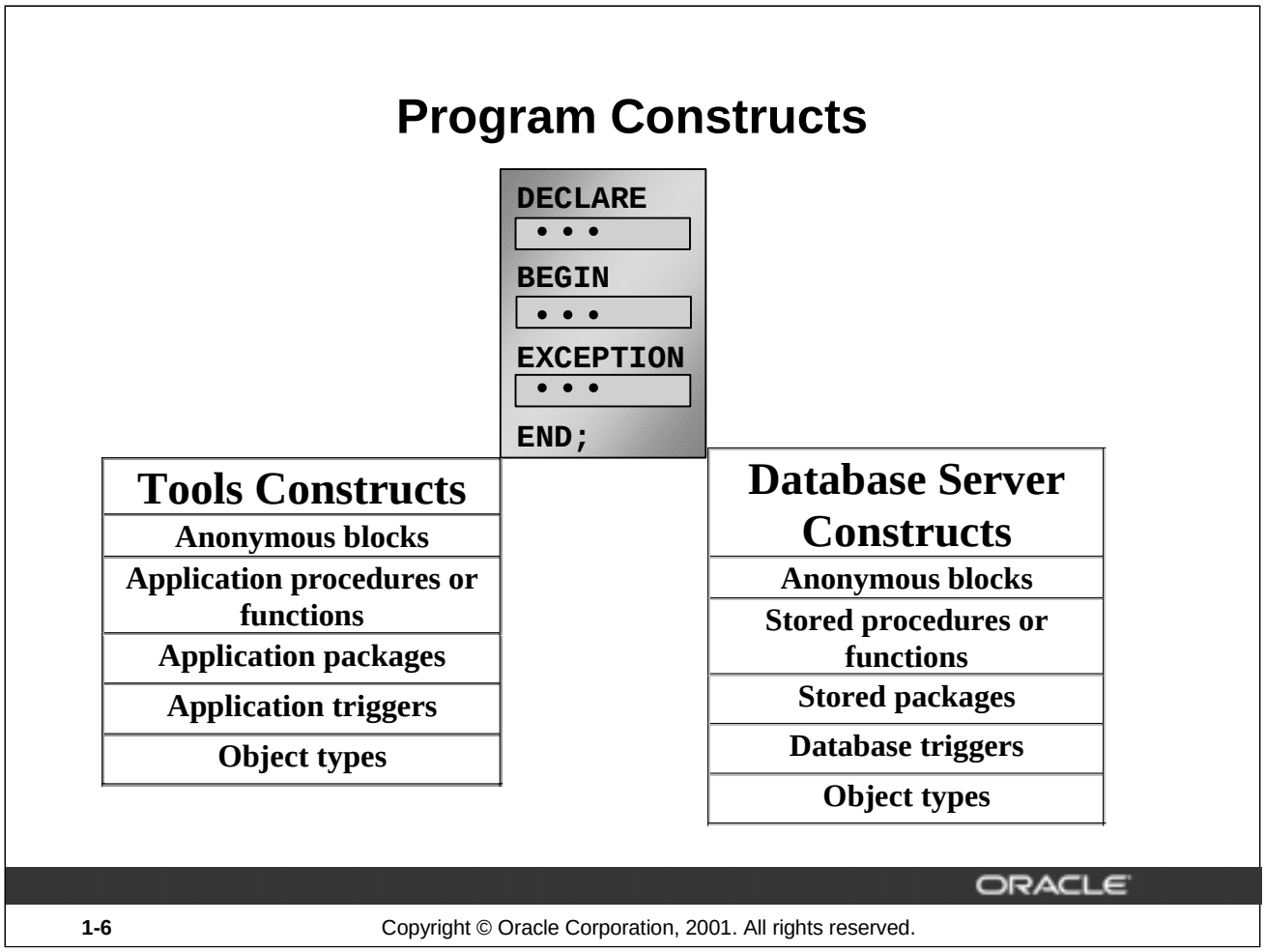
Anonymous blocks are unnamed blocks. They are declared at the point in an application where they are to be executed and are passed to the PL/SQL engine for execution at run time. You can embed an anonymous block within a precompiler program and within *i*SQL*Plus or Server Manager. Triggers in Oracle Developer components consist of such blocks.

Subprograms:

Subprograms are named PL/SQL blocks that can accept parameters and can be invoked. You can declare them either as procedures or as functions. Generally use a procedure to perform an action and a function to compute a value.

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

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



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
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
Database triggers	PL/SQL blocks that are associated with a database table and fired automatically when triggered by DML statements	Oracle server
Application triggers	PL/SQL blocks that are associated with an application event and fired automatically	Oracle Developer tools components, for example, Oracle Forms Developer
Object types	User-defined composite data types that encapsulates a data structure along with the functions and procedures needed to manipulate the data	Oracle server and Oracle Developer tools

Use of Variables

Variables can be used for:

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

ORACLE

Use of Variables

With PL/SQL you can declare variables and then use them in SQL and procedural statements anywhere that an expression can be used. Variables can be used for the following:

- Temporary storage of data: Data can be temporarily stored in one or more variables for use when validating data input and for processing later in the data flow process.
- Manipulation of stored values: Variables can be used for calculations and other data manipulations without accessing the database.
- Reusability: After they are declared, variables can be used repeatedly in an application simply by referencing them in other statements, including other declarative statements.
- Ease of maintenance: When using `%TYPE` and `%ROWTYPE` (more information on `%ROWTYPE` is covered in a subsequent lesson), you declare variables, basing the declarations on the definitions of database columns. If an underlying definition changes, the variable declaration changes accordingly at run time. This provides data independence, reduces maintenance costs, and allows programs to adapt as the database changes to meet new business needs. More information on `%TYPE` is covered later in this lesson.

Handling Variables in PL/SQL

- Declare and initialize variables in the declaration section.
- Assign new values to variables in the executable section.
- Pass values into PL/SQL blocks through parameters.
- View results through output variables.

ORACLE

Handling Variables in PL/SQL

Declare and Initialize Variables 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.

Assign New Values to Variables in the Executable Section

In the executable section, the existing value of the variable is replaced with the new value that is assigned to the variable.

Pass Values Into PL/SQL Subprograms Through Parameters

There are three parameter modes, `IN` (the default), `OUT`, and `IN OUT`. Use the IN parameter to pass values to the subprogram being called. Use the `OUT` parameter to return values to the caller of a subprogram. And use the `IN OUT` parameter to pass initial values to the subprogram being called and to return updated values to the caller. We pass values into anonymous block via *i*SQL*PLUS substitution variables.

Note: Viewing the results from a PL/SQL block through output variables is discussed later in the lesson.

Types of Variables

- **PL/SQL variables:**
  - **Scalar**
  - **Composite**
  - **Reference**
  - **LOB (large objects)**
- **Non-PL/SQL variables: Bind and host variables**

ORACLE

Types of Variables

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

- Scalar data types hold a single value. The main data types are those that correspond to column types in Oracle server tables; PL/SQL also supports Boolean variables.
- Composite data types, such as records, allow groups of fields to be defined and manipulated in PL/SQL blocks.
- Reference data types hold values, called pointers, that designate other program items. Reference data types are not covered in this course.
- LOB data types hold values, called locators, that specify the location of large objects (for example graphic images) that are stored out of line. LOB data types are discussed in detail later in this course.

Non-PL/SQL variables include host language variables declared in precompiler programs, screen fields in Forms applications, and *i*SQL*Plus host variables.

For more information on LOBs, see *PL/SQL User's Guide and Reference,* "Fundamentals."

Using *i*SQL*Plus Variables Within PL/SQL Blocks

- **PL/SQL does not have input or output capability of its own.**
- **You can reference substitution variables within a PL/SQL block with a preceding ampersand.**
- ***i*SQL*Plus host (or "bind") variables can be used to pass run time values out of the PL/SQL block back to the *i*SQL*Plus environment.**

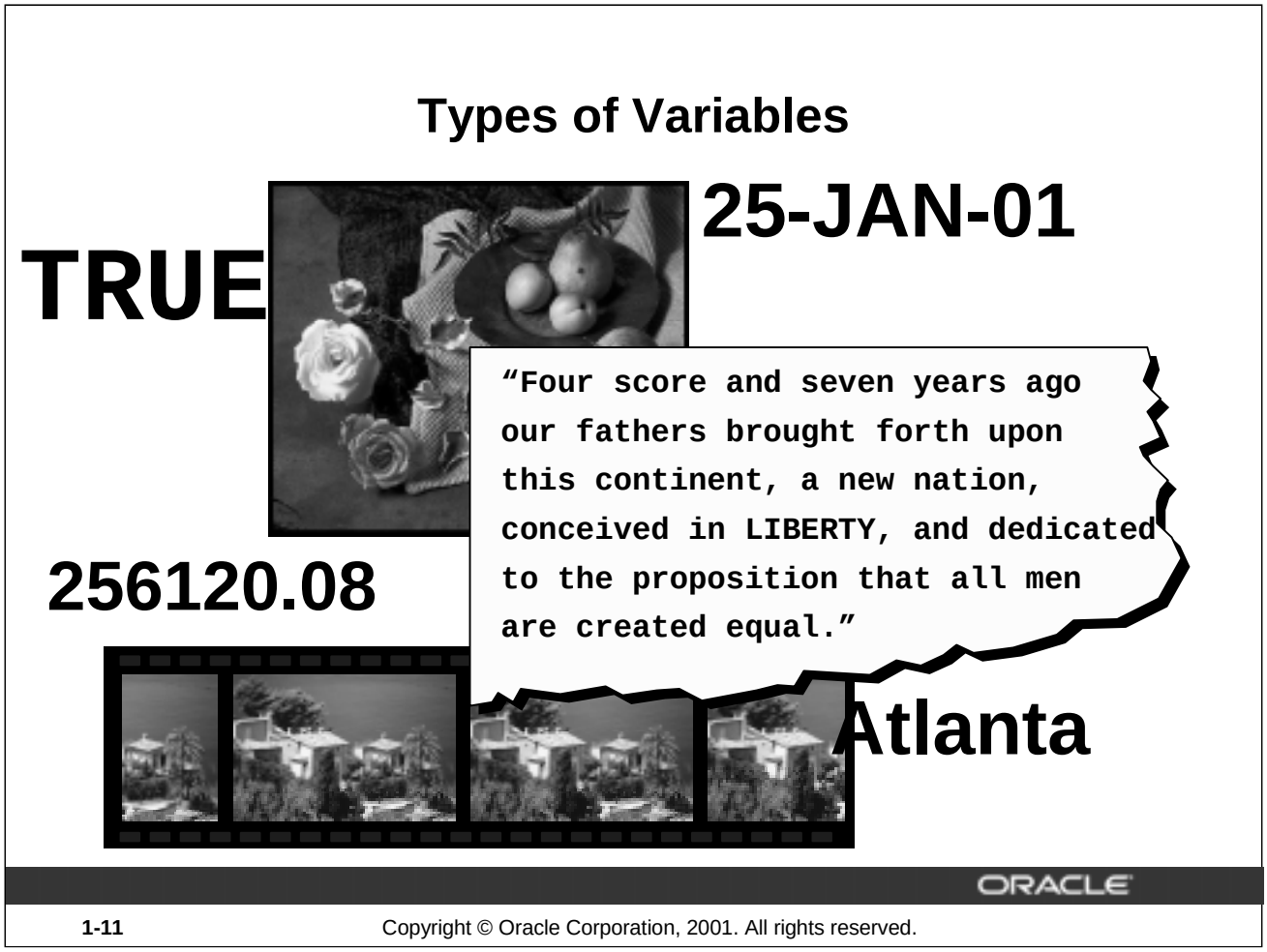
ORACLE

Using *i*SQL*Plus Variables Within PL/SQL Blocks

PL/SQL does not have input or output capability of its own. You must rely on the environment in which PL/SQL is executing to pass values into and out of a PL/SQL block.

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 within a PL/SQL block with a preceding ampersand in the same manner as you reference *i*SQL*Plus substitution variables in a SQL statement. 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. Only one value will replace the substitution variable.

*i*SQL*Plus host variables can be used to pass run time values out of the PL/SQL block back to the *i*SQL*Plus environment. You can reference host variables in a PL/SQL block with a preceding colon. Bind variables are discussed in further detail later in this lesson.



Types of Variables

The slide illustrates the following variable data types:

- TRUE represents a Boolean value.
- 25-JAN-01 represents a DATE.
- The photograph represents a BLOB.
- The text of a speech represents a LONG.
- 256120.08 represents a NUMBER data type with precision and scale.
- The movie represents a BFILE.
- The city name, Atlanta, represents a VARCHAR2.

Declaring PL/SQL Variables

Syntax:

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

Examples:

```
DECLARE
  v_hiredate      DATE;
  v_deptno        NUMBER(2) NOT NULL := 10;
  v_location      VARCHAR2(13) := 'Atlanta';
  c_comm          CONSTANT NUMBER := 1400;
```

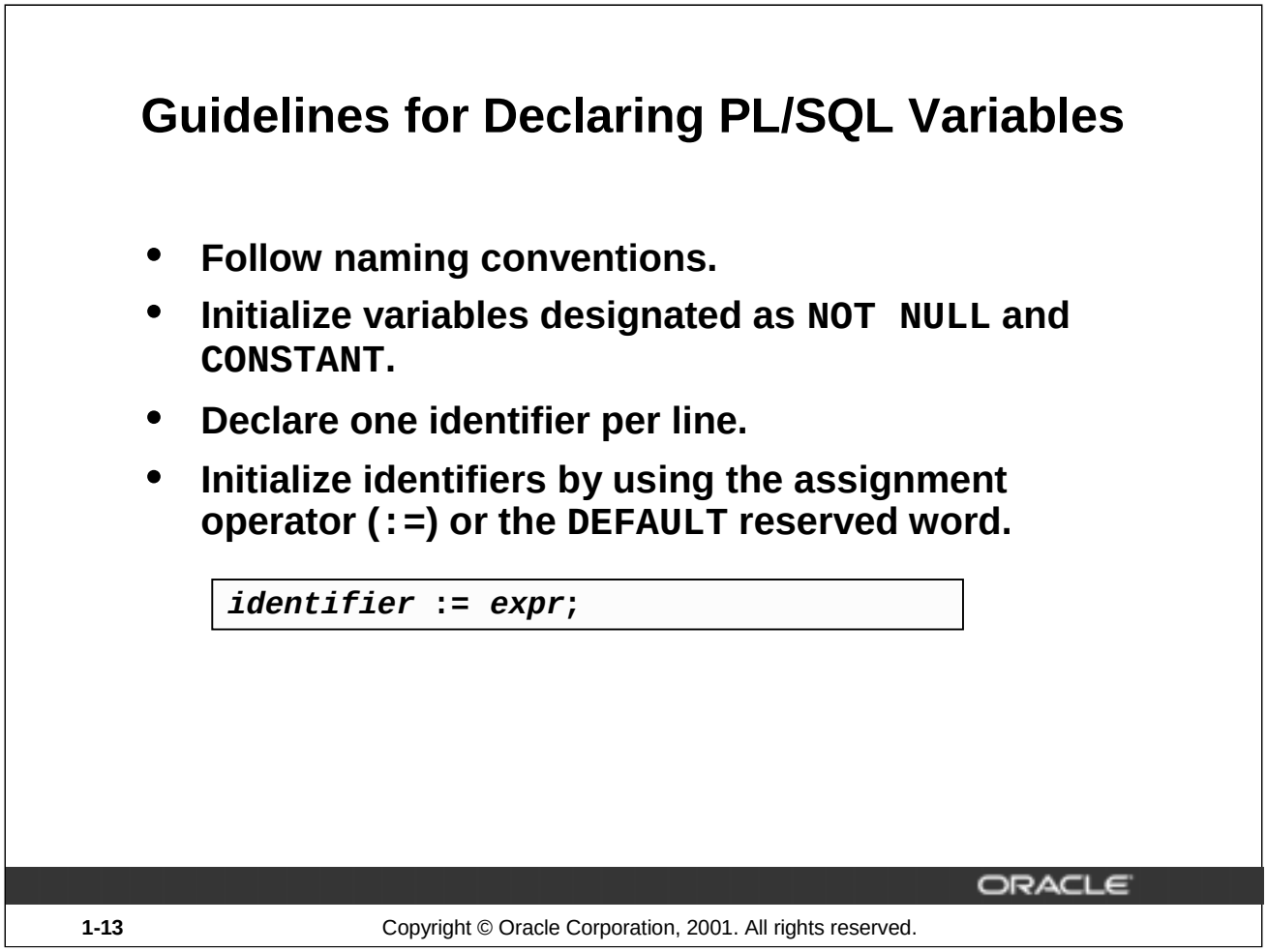
ORACLE

Declaring 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 to assign an initial value to a variable. 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.



Guidelines for Declaring PL/SQL Variables

Here are some guidelines to follow while declaring PL/SQL variables:

- Name the identifier according to the same rules used for SQL objects.
- You can use naming conventions—for example, *v_name* to represent a variable and *c_name* to represent a constant variable.
- If you use the `NOT NULL` constraint, you must assign a value.
- Declaring only one identifier per line makes code easier to read and maintain.
- 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 50000 to the constant. A constant must be initialized in its declaration; otherwise, you get a compilation error when the declaration is elaborated (compiled).

  ```
  v_sal     CONSTANT REAL := 50000.00;
  ```

- Initialize the variable to an expression with the assignment operator (`:=`) or, equivalently, 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 later. To assign or reassign a value to a variable, you write a PL/SQL assignment statement. You must explicitly name the variable to receive the new value to the left of the assignment operator (`:=`). It is good programming practice to initialize all variables.

Naming Rules

- **Two variables can have the same name, provided they are in different blocks.**
- **The variable name (identifier) should not be the same as the name of table columns used in the block.**

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

Adopt a naming convention for PL/SQL identifiers: for example, v_employee_id

ORACLE

Naming Rules

Two objects can have the same name, provided that they are defined in different blocks. Where they coexist, only the object declared in the current block can be used.

You should not choose the same name (identifier) for a variable as the name of table columns used in the block. 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.

Consider adopting a naming convention for various objects that are declared in the DECLARE section of the PL/SQL block. Using v_ as a prefix representing *variable* avoids naming conflicts with database objects.

```
DECLARE
   v_hire_date       date;
BEGIN
...
```

Note: The names of the variables must not be longer than 30 characters. The first character must be a letter; the remaining characters can be letters, numbers, or special symbols.

Variable Initialization and Keywords

- **Assignment operator (:=)**
- **DEFAULT keyword**
- **NOT NULL constraint**

Syntax

```
identifier := expr;
```

Examples:

```
v_hiredate := '01-JAN-2001';
```

```
v_ename := 'Maduro';
```

ORACLE

Variable Initialization and Keywords

In the syntax:

identifier is the name of the scalar variable.

expr can be a variable, literal, or function call, but *not* a database column.

The variable value assignment examples are defined as follows:

- Set the identifier V_HIREDATE to a value of 01-JAN-2001.
- Store the name "Maduro" in the V_ENAME identifier.

Variables are initialized every time a block or subprogram is entered. By default, variables are initialized to NULL. Unless you explicitly initialize a variable, its value is undefined.

Use the assignment operator (:=) for variables that have no typical value.

```
v_hire_date := '15-SEP-1999'
```

Note: This four-digit value for year, YYYY, assignment is possible only in Oracle8*i* and later. Previous versions may require the use of the TO_DATE function.

DEFAULT: You can use the DEFAULT keyword instead of the assignment operator to initialize variables. Use DEFAULT for variables that have a typical value.

```
v_mgr      NUMBER(6) DEFAULT 100;
```

NOT NULL: 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.

```
v_city     VARCHAR2(30) NOT NULL := 'Oxford'
```

Variable Initialization and Keywords (continued)

Note: String literals must be enclosed in single quotation marks. For example, `'Hello, world'`. If there is a single quotation mark in the string, use a single quotation mark twice—for example, to insert a value FISHERMAN'S DRIVE, the string would be `'FISHERMAN''S DRIVE'.`

Another way to assign values to variables is to select or fetch database values into it. The following example computes a 10% bonus for the employee with the `EMPLOYEE_ID` 176 and assigns the computed value to the `v_bonus` variable. This is done using the `INTO` clause.

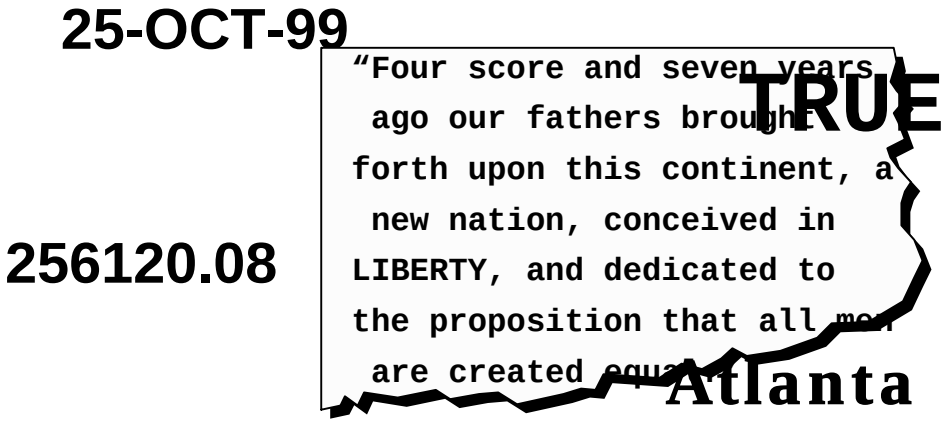
```
DECLARE
    v_bonus NUMBER(8,2);
BEGIN
    SELECT  salary * 0.10
    INTO    v_bonus
    FROM    employees
    WHERE   employee_id = 176;
END;
/
```

Then you can use the variable `v_bonus` in another computation or insert its value into a database table.

Note: To assign a value into a variable from the database, use a `SELECT` or `FETCH` statement. The `FETCH` statement is covered later in this course.

Scalar Data Types

- Hold a single value
- Have no internal components



ORACLE

Scalar Data Types

Every constant, variable, and parameter has a data type (or type), which 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, reference, and LOB types. In addition, This chapter covers the basic types that are used frequently in PL/SQL programs. Later chapters cover the more specialized types.

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 *PL/SQL User's Guide and Reference*, "Fundamentals."

Base Scalar Data Types

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

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.
LONG	Base type for variable-length character data up to 32,760 bytes. Use the LONG data type to store variable-length character strings. You can insert any LONG value into a LONG database column because the maximum width of a LONG column is 2**31 bytes. However, you cannot retrieve a value longer than 32760 bytes from a LONG column into a LONG variable.
LONG RAW	Base type for binary data and byte strings up to 32,760 bytes. LONG RAW data is not interpreted by PL/SQL.
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.

Base Scalar Data types (continued)

Data Type	Description
BINARY_INTEGER	Base type for integers between -2,147,483,647 and 2,147,483,647.
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 and BINARY_INTEGER values.
BOOLEAN	Base type that stores one of three possible values used for logical calculations: TRUE, FALSE, or NULL.

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, and 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 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, Oracle 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 years_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.

Scalar Variable Declarations

Examples:

```
DECLARE
  v_job          VARCHAR2(9);
  v_count        BINARY_INTEGER := 0;
  v_total_sal    NUMBER(9,2) := 0;
  v_orderdate    DATE := SYSDATE + 7;
  c_tax_rate     CONSTANT NUMBER(3,2) := 8.25;
  v_valid        BOOLEAN NOT NULL := TRUE;
  ...
```

ORACLE

Declaring Scalar Variables

The examples of variable declaration shown on the slide are defined as follows:

- `v_job`: variable to store an employee job title
- `v_count`: variable to count the iterations of a loop and initialized to 0
- `v_total_sal`: variable to accumulate the total salary for a department and initialized to 0
- `v_orderdate`: variable to store the ship date of an order and initialize to one week from today
- `c_tax_rate`: a constant variable for the tax rate, which never changes throughout the PL/SQL block
- `v_valid`: flag to indicate whether a piece of data is valid or invalid and initialized to `TRUE`

The %TYPE Attribute

- **Declare a variable according to:**
  - **A database column definition**
  - **Another previously declared variable**
- **Prefix %TYPE with:**
  - **The database table and column**
  - **The previously declared variable name**

ORACLE

The %TYPE Attribute

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 will occur during execution.

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 will be derived from a table in the database. To use the attribute in place of the data type that is required in the variable declaration, prefix it with the database table and column name. If referring to a previously declared variable, prefix the variable name to the attribute.

PL/SQL determines the data type and size of the variable when the block is compiled so that such variables are always compatible with the column that is used to populate it. This is a definite advantage for writing and maintaining code, because there is no need to be concerned with column data type changes made at the database level. You can also declare a variable according to another previously declared variable by prefixing the variable name to the attribute.

Declaring Variables with the %TYPE Attribute

Syntax:

```
identifier     Table.column_name%TYPE;
```

Examples:

```
...
  v_name            employees.last_name%TYPE;
  v_balance         NUMBER(7,2);
  v_min_balance     v_balance%TYPE := 10;
...
```

ORACLE

Declaring Variables with the %TYPE Attribute

Declare variables to store the last name of an employee. The variable v_name is defined to be of the same data type as the LAST_NAME column in the EMPLOYEES table. %TYPE provides the data type of a database column:

```
...
v_name      employees.last_name%TYPE;
...
```

Declare variables to store the balance of a bank account, as well as the minimum balance, which starts out as 10. The variable v_min_balance is defined to be of the same data type as the variable v_balance. %TYPE provides the data type of a variable:

```
...
v_balance      NUMBER(7,2);
v_min_balance  v_balance%TYPE := 10;
...
```

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.**
- **The variables are compared by the logical operators `AND`, `OR`, and `NOT`.**
- **The variables always yield `TRUE`, `FALSE`, or `NULL`.**
- **Arithmetic, character, and date expressions can be used to return a Boolean value.**

ORACLE

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

```
v_sal1 := 50000;
v_sal2 := 60000;
```

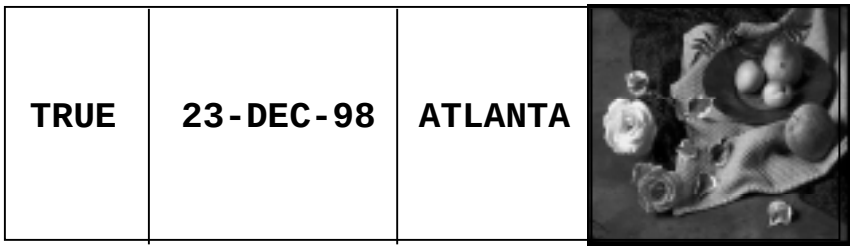
The following expression yields `TRUE`:

```
v_sal1 < v_sal2
```

Declare and initialize a Boolean variable:

```
DECLARE
    v_flag BOOLEAN := FALSE;
BEGIN
    v_flag := TRUE;
END;
```

Composite Data Types



PL/SQL table structure

1	SMITH
2	JONES
3	NANCY
4	TIM

VARCHAR2

BINARY_INTEGER

PL/SQL table structure

1	5000
2	2345
3	12
4	3456

NUMBER

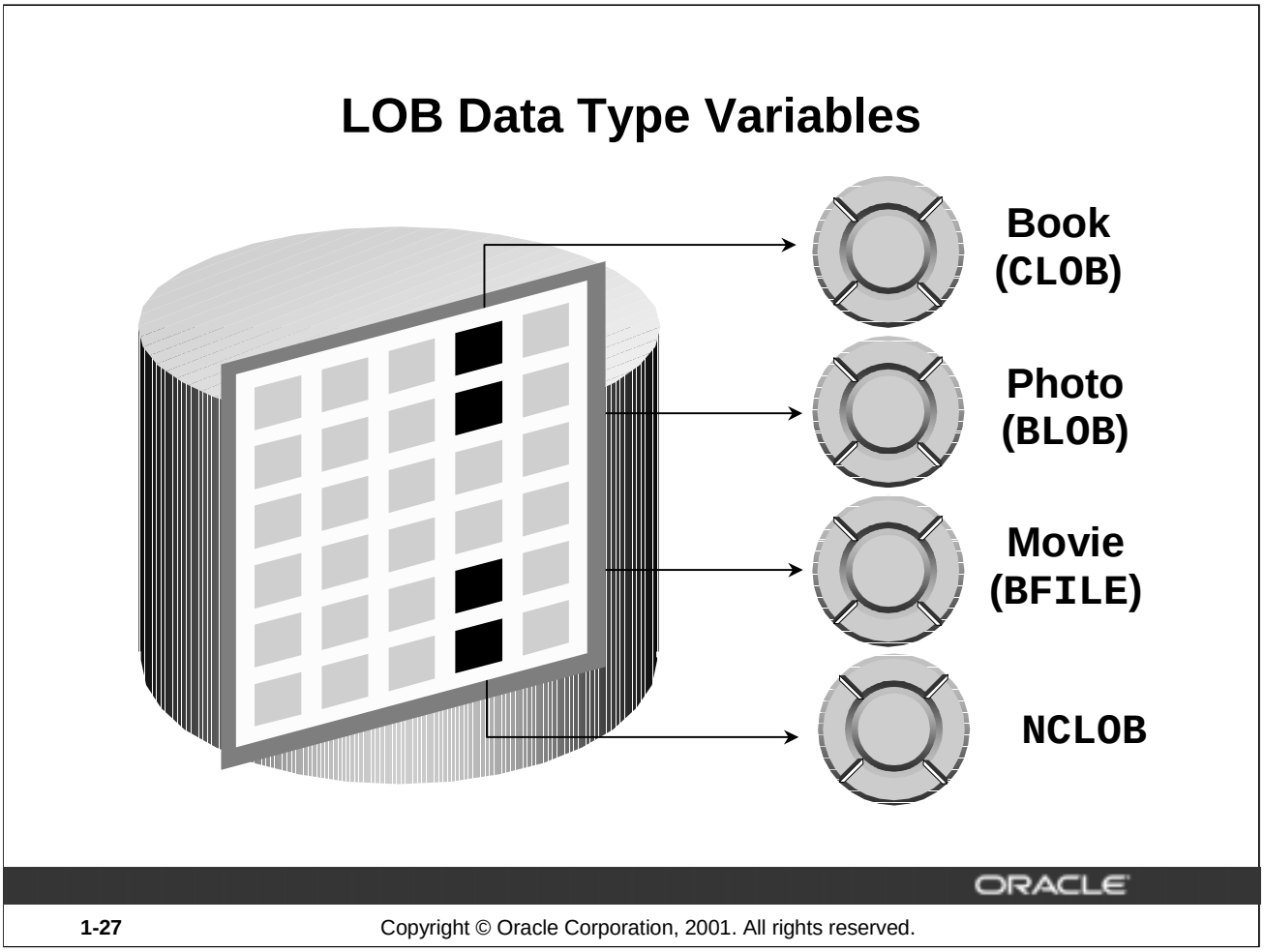
BINARY_INTEGER

ORACLE

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 RECORD data type to treat related but dissimilar data as a logical unit. Use the TABLE data type to reference and manipulate collections of data as a whole object. Both RECORD and TABLE data types are covered in detail in a subsequent lesson. NESTED TABLE and VARRAY data types are covered in the *Advanced PL/SQL* course.

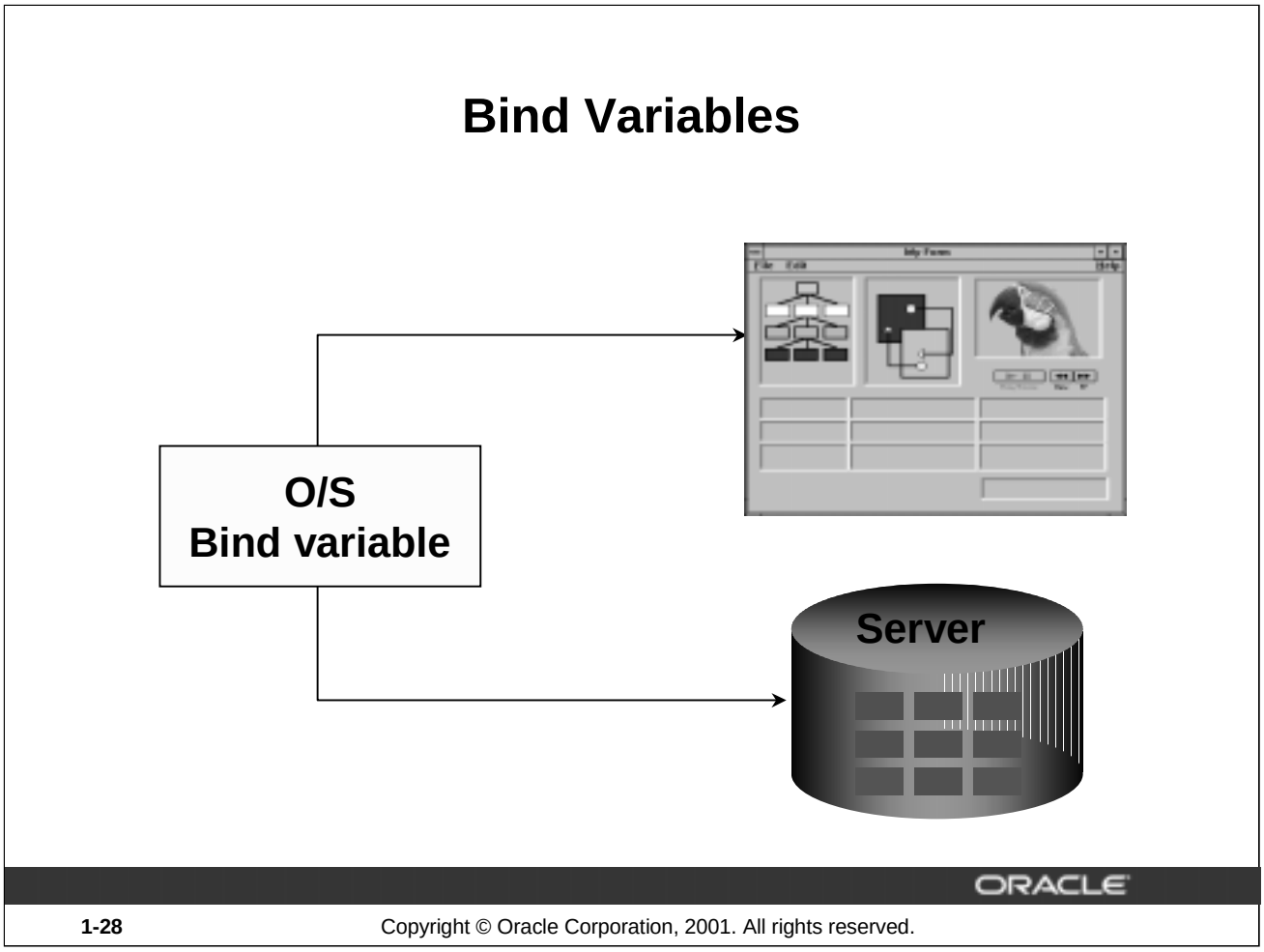
For more information, see *PL/SQL User's Guide and Reference*, "Collections and Records."



LOB Data Type Variables

With the LOB (large object) data types you can store blocks of unstructured data (such as text, graphic images, video clips, and sound wave forms) up to 4 gigabytes in size. LOB data types allow efficient, random, piecewise access to the data and can be attributes of an object type. LOBs also support random access to data.

- The CLOB (character large object) data type is used to store large blocks of single-byte character data in the database in line (inside the row) or out of line (outside the row).
- The BLOB (binary large object) data type is used to store large binary objects in the database in line (inside the row) or out of line (outside the row).
- The BFILE (binary file) data type is used to store large binary objects in operating system files outside the database.
- The NCLOB (national language character large object) data type is used to store large blocks of single-byte or fixed-width multibyte NCHAR unicode data in the database, in line or out of line.



Bind Variables

A bind variable is a variable that you declare in a host environment. Bind variables can be used to pass run-time values, either number or character, into or out of one or more PL/SQL programs. The PL/SQL programs use bind variables as they would use any other variable. You can reference variables declared in the host or calling environment in PL/SQL statements, unless the statement is in a procedure, function, or package. This includes host language variables declared in precompiler programs, screen fields in Oracle Developer Forms applications, and *i*SQL*Plus bind variables.

Creating Bind Variables

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

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

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

Displaying Bind Variables

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. The following example illustrates a PRINT command:

```
VARIABLE g_n NUMBER
...
PRINT g_n
```

You can reference host variables in PL/SQL programs. These variables should be preceded by a colon.

```
VARIABLE RESULT NUMBER
```

An example of using a host variable in a PL/SQL block:

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

Using Bind Variables

To reference a bind variable in PL/SQL, you must prefix its name with a colon (:).

Example:

```
VARIABLE      g_salary NUMBER
BEGIN
   SELECT     salary
   INTO        :g_salary
   FROM       employees
   WHERE      employee_id = 178;
END;
/
PRINT g_salary
```

ORACLE

Printing Bind Variables

In *i*SQL*Plus you can display the value of the bind variable using the `PRINT` command.

G_SALARY
7000

Referencing Non-PL/SQL Variables

Store the annual salary into a *i*SQL*Plus host variable.

```
:g_monthly_sal := v_sal / 12;
```

- **Reference non-PL/SQL variables as host variables.**
- **Prefix the references with a colon (:).**

Referencing Non-PL/SQL Variables

To reference host variables, you must prefix the references with a colon (:) to distinguish them from declared PL/SQL variables.

Example

This example computes the monthly salary, based upon the annual salary supplied by the user. This script contains both *i*SQL*Plus commands as well as a complete PL/SQL block.

```
SET VERIFY OFF
VARIABLE    g_monthly_sal  NUMBER
DEFINE     p_annual_sal = 50000

DECLARE
  v_sal     NUMBER(9,2) := &p_annual_sal;
BEGIN
  :g_monthly_sal := v_sal/12;
END;
/
PRINT       g_monthly_sal
```

The `DEFINE` command specifies a user variable and assigns it a `CHAR` value. Even though you enter the number 50000, *i*SQL*Plus assigns a `CHAR` value to `p_annual_sal` consisting of the characters, 5,0,0,0 and 0.

DBMS_OUTPUT.PUT_LINE

- An Oracle-supplied packaged procedure
- An alternative for displaying data from a PL/SQL block
- Must be enabled in *i*SQL*Plus with SET SERVEROUTPUT ON

```
SET SERVEROUTPUT ON
DEFINE p_annual_sal = 60000
DECLARE
  v_sal NUMBER(9,2) := &p_annual_sal;
BEGIN
  v_sal := v_sal/12;
  DBMS_OUTPUT.PUT_LINE ('The monthly salary is ' ||
                        TO_CHAR(v_sal));
END;
/
```

ORACLE

DBMS_OUTPUT.PUT_LINE

You have seen that you can declare a host variable, reference it in a PL/SQL block, and then display its contents in *i*SQL*Plus using the `PRINT` command. Another option for displaying information from a PL/SQL block is `DBMS_OUTPUT.PUT_LINE`. `DBMS_OUTPUT` is an Oracle-supplied package, and `PUT_LINE` is a procedure within that package.

Within a PL/SQL block, reference `DBMS_OUTPUT.PUT_LINE` and, in parentheses, specify the string that you want to print to the screen. The package must first be enabled in your *i*SQL*Plus session. To do this, execute the *i*SQL*Plus `SET SERVEROUTPUT ON` command.

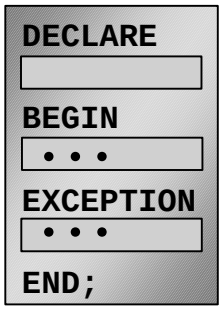
The example on the slide computes the monthly salary and prints it to the screen, using `DBMS_OUTPUT.PUT_LINE`. The output is shown below:

```
The monthly salary is 5000
PL/SQL procedure successfully completed.
```

Summary

In this lesson you should have learned the following:

- **PL/SQL blocks are composed of the following sections:**
  - **Declarative (optional)**
  - **Executable (required)**
  - **Exception handling (optional)**
- **A PL/SQL block can be an anonymous block, procedure, or function.**



ORACLE

Summary

A PL/SQL block is a basic, unnamed unit of a PL/SQL program. It consists of a set of SQL or PL/SQL statements and it performs a single 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. The executable part is the mandatory part of a PL/SQL block, and contains SQL and PL/SQL statements for querying and manipulating data. The exception-handling part is embedded inside the executable part of a block and is placed at the end of the executable part.

An anonymous PL/SQL block is the basic, unnamed unit of a PL/SQL program. Procedures and functions can be compiled separately and stored permanently in an Oracle database, ready to be executed.

Summary

In this lesson you should have learned the following:

- **PL/SQL identifiers:**
  - **Are defined in the declarative section**
  - **Can be of scalar, composite, reference, or `LOB` data type**
  - **Can be based on the structure of another variable or database object**
  - **Can be initialized**
- **Variables declared in an external environment such as *i*SQL*Plus are called host variables.**
- **Use `DBMS_OUTPUT.PUT_LINE` to display data from a PL/SQL block.**

ORACLE

Summary (continued)

All PL/SQL data types are scalar, composite, reference, or `LOB` type. Scalar data types do not have any components within them, whereas composite data types have other data types within them. PL/SQL variables are declared and initialized in the declarative section.

When a PL/SQL program is written and executed using *i*SQL*Plus, *i*SQL*Plus becomes the host environment for the PL/SQL program. The variables declared in *i*SQL*Plus are called host variables. Then the PL/SQL program is written and executed using, for example, Oracle Forms. Forms becomes a host environment, and variables declared in Oracle Forms are called host variables. Host variables are also called bind variables.

To display information from a PL/SQL block use `DBMS_OUTPUT.PUT_LINE`.`DBMS_OUTPUT` is an Oracle-supplied package, and `PUT_LINE` is a procedure within that package. Within a PL/SQL block, reference `DBMS_OUTPUT.PUT_LINE` and, in parentheses, specify the string that you want to print to the screen.

Practice 1 Overview

This practice covers the following topics:

- **Determining validity of declarations**
- **Declaring a simple PL/SQL block**
- **Executing a simple PL/SQL block**

ORACLE

Practice 1 Overview

This practice reinforces the basics of PL/SQL covered in this lesson, including data types, definitions of identifiers, and validation of expressions. You put all these elements together to create a simple PL/SQL block.

Paper-Based Questions

Questions 1 and 2 are paper-based questions.

Practice 1

1. Evaluate each of the following declarations. Determine which of them are *not* legal and explain why.

   a.
   ```
   DECLARE
     v_id                    NUMBER(4);
   ```

   b.
   ```
   DECLARE
     v_x, v_y, v_z   VARCHAR2(10);
   ```

   c.
   ```
   DECLARE
     v_birthdate             DATE NOT NULL;
   ```

   d.
   ```
   DECLARE
     v_in_stock              BOOLEAN := 1;
   ```

Practice 1 (continued)

2. In each of the following assignments, indicate whether the statement is valid and what the valid data type of the result will be.

   a. `v_days_to_go := v_due_date - SYSDATE;`

   b. `v_sender := USER || ': ' || TO_CHAR(v_dept_no);`

   c. `v_sum := $100,000 + $250,000;`

   d. `v_flag := TRUE;`

   e. `v_n1 := v_n2 > (2 * v_n3);`

   f. `v_value := NULL;`

3. Create an anonymous block to output the phrase "My PL/SQL Block Works" to the screen.

G_MESSAGE
My PL/SQL Block Works

Practice 1 (continued)

If you have time, complete the following exercise:

4. Create a block that declares two variables. Assign the value of these PL/SQL variables to *i*SQL*Plus host variables and print the results of the PL/SQL variables to the screen. Execute your PL/SQL block. Save your PL/SQL block in a file named `p1q4.sql`, by clicking the `Save Script` button. Remember to save the script with a `.sql` extension.

```
V_CHAR    Character (variable length)
V_NUM     Number
```

Assign values to these variables as follows:

```
Variable Value
-------- -------------------------------------
V_CHAR   The literal '42 is the answer'
V_NUM    The first two characters from V_CHAR
```

G_CHAR
42 is the answer

G_NUM
42

2

Writing Executable Statements

ORACLE

Objectives

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

- **Describe the significance of the executable section**
- **Use identifiers correctly**
- **Write statements in the executable section**
- **Describe the rules of nested blocks**
- **Execute and test a PL/SQL block**
- **Use coding conventions**

ORACLE

Lesson Aim

In this lesson, you learn how to write executable code in the PL/SQL block. You also learn the rules for nesting PL/SQL blocks of code, as well as how to execute and test PL/SQL code.

PL/SQL Block Syntax and Guidelines

- **Statements can continue over several lines.**
- **Lexical units can be classified as:**
  - **Delimiters**
  - **Identifiers**
  - **Literals**
  - **Comments**

ORACLE

PL/SQL Block Syntax and Guidelines

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

- A line of PL/SQL text contains groups of characters known as lexical units, which can be classified as follows:
  - Delimiters (simple and compound symbols)
  - Identifiers, which include reserved words
  - Literals
  - Comments
- To improve readability, you can separate lexical units by spaces. In fact, you must separate adjacent identifiers by a space or punctuation.
- You cannot embed spaces in lexical units except for string literals and comments.
- Statements can be split across lines, but keywords must not be split.

PL/SQL Block Syntax and Guidelines (continued)

Delimiters

Delimiters are simple or compound symbols that have special meaning to PL/SQL.

Simple Symbols

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

Compound Symbols

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

Identifiers

- Can contain up to 30 characters
- Must begin with an alphabetic character
- Can contain numerals, dollar signs, underscores, and number signs
- Can not contain characters such as hyphens, slashes, and spaces
- Should not have the same name as a database table column name
- Should not be reserved words

ORACLE

Identifiers

Identifiers are used to name PL/SQL program items and units, which include constants, variables, exceptions, cursors, cursor variables, subprograms, and packages.

- Identifiers can contain up to 30 characters, but they must start with an alphabetic character.
- Do not choose the same name for the identifier as the name of columns in a table used in the block. If PL/SQL identifiers are in the same SQL statements and have the same name as a column, then Oracle assumes that it is the column that is being referenced.
- Reserved words should be written in uppercase to promote readability.
- An identifier consists of a letter, optionally followed by more letters, numerals, dollar signs, underscores, and number signs. Other characters such as hyphens, slashes, and spaces are illegal, as the following examples show:

```
dots&dashes   -- illegal ampersand
debit-amount  -- illegal hyphen
on/off        -- illegal slash
user id       -- illegal space
```

- The next examples show that adjoining and trailing dollar signs, underscores, and number signs are allowed:

```
money$$$tree
SN##
try_again_
```

Note: Reserved words cannot be used as identifiers unless they are enclosed in double quotation marks (for example, "SELECT").

PL/SQL Block Syntax and Guidelines

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

```
v_name := 'Henderson';
```

  - **Numbers can be simple values or scientific notation.**
- **A slash (/) runs the PL/SQL block in a script file or in some tools such as *i*SQL*PLUS.**

ORACLE

PL/SQL Block Syntax and Guidelines

A literal is an explicit numeric, character, string, 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 by a scientific notation (for example, `2E5`, meaning `2* (10 to the power of 5) = 200000`).

A PL/SQL program is terminated and executed by a slash (`/`) on a line by itself.

Commenting Code

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

Example:

```
DECLARE
...
  v_sal NUMBER (9,2);
BEGIN
  /* Compute the annual salary based on the
     monthly salary input from the user */
  v_sal := :g_monthly_sal * 12;
END;     -- This is the end of the block
```

ORACLE

Commenting Code

Comment code to document each phase and to assist debugging. Comment the PL/SQL code with two dashes (--) 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 behavioral logic or data. Well-placed comments are extremely valuable for code readability and future code maintenance.

Example

In the example on the slide, the line enclosed within /* and */ is the comment that explains the code that follows it.

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**

  } **Same as in SQL**

- **Not available in procedural statements:**
  - **DECODE**
  - **Group functions**

SQL Functions in PL/SQL

Most of the functions available in SQL are also valid in PL/SQL expressions:

- Single-row number functions
- Single-row character functions
- Data type conversion functions
- Date functions
- Timestamp functions
- `GREATEST`, `LEAST`
- Miscellaneous functions

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.

SQL Functions in PL/SQL: Examples

- **Build the mailing list for a company.**

```
v_mailing_address := v_name||CHR(10)||
                     v_address||CHR(10)||v_state||
                     CHR(10)||v_zip;
```

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

```
v_ename       := LOWER(v_ename);
```

ORACLE

SQL Functions in PL/SQL: Examples

Most of the SQL functions can be used in PL/SQL. These built-in functions help you to manipulate data; they fall into the following categories:

- Number
- Character
- Conversion
- Date
- Miscellaneous

The function examples in the slide are defined as follows:

- Build the mailing address for a company.
- Convert the name to lowercase.

`CHR` is the SQL function that converts an ASCII code to its corresponding character; 10 is the code for a line feed.

PL/SQL has its own error handling functions which are:

- `SQLCODE`
- `SQLERRM`

These functions are discussed later in this course.

For more information, see *PL/SQL User's Guide and Reference*, "Fundamentals."

Data type Conversion

- **Convert data to comparable data types.**
- **Mixed data types can result in an error and affect performance.**
- **Conversion functions:**
  - **TO_CHAR**
  - **TO_DATE**
  - **TO_NUMBER**

```
DECLARE
   v_date DATE := TO_DATE('12-JAN-2001', 'DD-MON-YYYY');
BEGIN
   . . .
```

ORACLE

Data type Conversion

PL/SQL attempts to convert data types dynamically if they are mixed in a statement. For example, if you assign a NUMBER value to a CHAR variable, then PL/SQL dynamically translates the number into a character representation, so that it can be stored in the CHAR variable. The reverse situation also applies, provided that the character expression represents a numeric value.

If they are compatible, you can also assign characters to DATE variables and vice versa.

Within an expression, you should make sure that data types are the same. If mixed data types occur in an expression, you should use the appropriate conversion function to convert the data.

Syntax

```
TO_CHAR (value, fmt)
TO_DATE (value, fmt)
TO_NUMBER (value, fmt)
```

where: *value* is a character string, number, or date.

fmt is the format model used to convert a value.

Data type Conversion

1. **This statement produces a compilation error if the variable `v_date` is declared as a `DATE` data type.**

```
v_date := 'January 13, 2001';
```

2. **To correct the error, use the `TO_DATE` conversion function.**

```
v_date := TO_DATE ('January 13, 2001',
                   'Month DD, YYYY');
```

ORACLE

Data type Conversion

The conversion examples in the slide are defined as follows:

1. Store a character string representing a date in a variable that is declared as a `DATE` data type. *This code causes a syntax error.*
2. To correct the error, convert the string to a date with the `TO_DATE` conversion function.

PL/SQL attempts conversion if possible, but its success depends on the operations that are being performed. It is good programming practice to explicitly perform data type conversions, because they can favorably affect performance and remain valid even with a change in software versions.

Nested Blocks and Variable Scope

- **PL/SQL blocks can be nested wherever an executable statement is allowed.**
- **A nested block becomes a statement.**
- **An exception section can contain nested blocks.**
- **The scope of an identifier is that region of a program unit (block, subprogram, or package) from which you can reference the identifier.**

ORACLE

Nested Blocks

One of the advantages that PL/SQL has over SQL is the ability to nest statements. You can nest blocks wherever an executable statement is allowed, thus making the nested block a statement. Therefore, you can break down the executable part of a block into smaller blocks. The exception section can also contain nested blocks.

Variable Scope

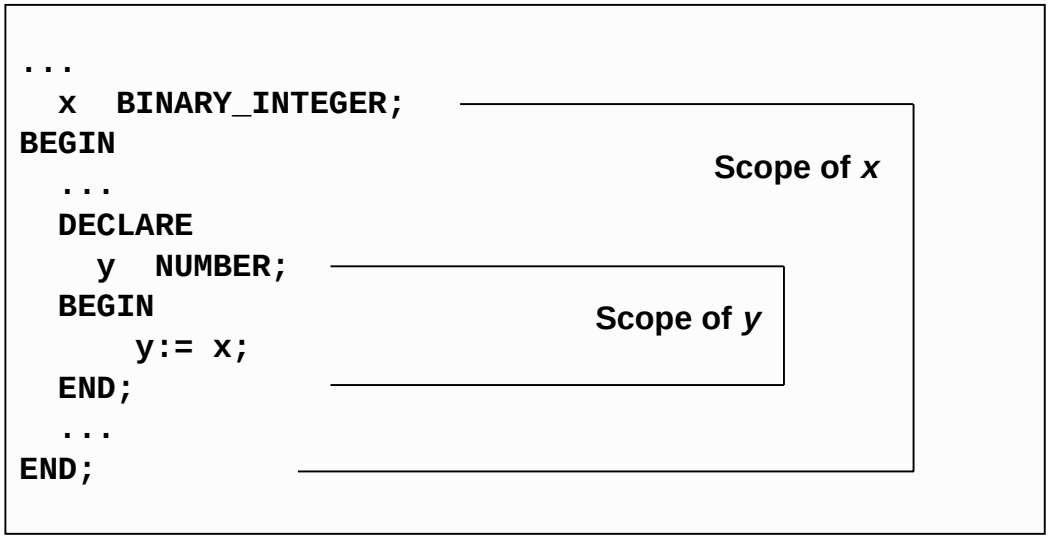
References to an identifier are resolved according to its scope and visibility. The scope of an identifier is that region of a program unit (block, subprogram, or package) from which you can reference the identifier. An identifier is visible only in the regions from which you can reference the identifier using an unqualified name.

Identifiers declared in a PL/SQL block are considered local to that block and global to all its subblocks. If a global identifier is redeclared in a subblock, both identifiers remain in scope. Within the subblock, however, only the local identifier is visible because you must use a qualified name to reference the global identifier.

Although you cannot declare an identifier twice in the same block, you can declare the same identifier in two different blocks. The two items represented by the identifier are distinct, and any change in one does not affect the other. However, a block cannot reference identifiers declared in other blocks at the same level because those identifiers are neither local nor global to the block.

Nested Blocks and Variable Scope

Example:



ORACLE

Nested Blocks and Variable Scope

In the nested block shown on the slide, the variable named *y* can reference the variable named *x*. Variable *x*, however, cannot reference variable *y*. If variable *y* in the nested block is given the same name as variable *x* in the outer block, its value is valid only for the duration of the nested block.

Scope

The scope of an identifier is that region of a program unit (block, subprogram, or package) from which you can reference the identifier.

Visibility

An identifier is visible only in the regions from which you can reference the identifier using an unqualified name.

Identifier Scope

An identifier is visible in the regions where you can reference the identifier without having to qualify it:

- **A block can look up to the enclosing block.**
- **A block cannot look down to enclosed blocks.**

ORACLE

Identifier Scope

An identifier is visible in the block in which it is declared and in all nested subblocks, procedures, and functions. If the block does not find the identifier declared locally, it looks *up* to the declarative section of the enclosing (or parent) blocks. The block never looks *down* to enclosed (or child) blocks or sideways to sibling blocks.

Scope applies to all declared objects, including variables, cursors, user-defined exceptions, and constants.

Qualify an Identifier

- The qualifier can be the label of an enclosing block.
- Qualify an identifier by using the block label prefix.

```
<<outer>>
   DECLARE
      birthdate DATE;
   BEGIN
         DECLARE
             birthdate DATE;
         BEGIN
            ...
            outer.birthdate :=
                           TO_DATE('03-AUG-1976',
                                    'DD-MON-YYYY');
         END;
    ....
    END;
```

ORACLE

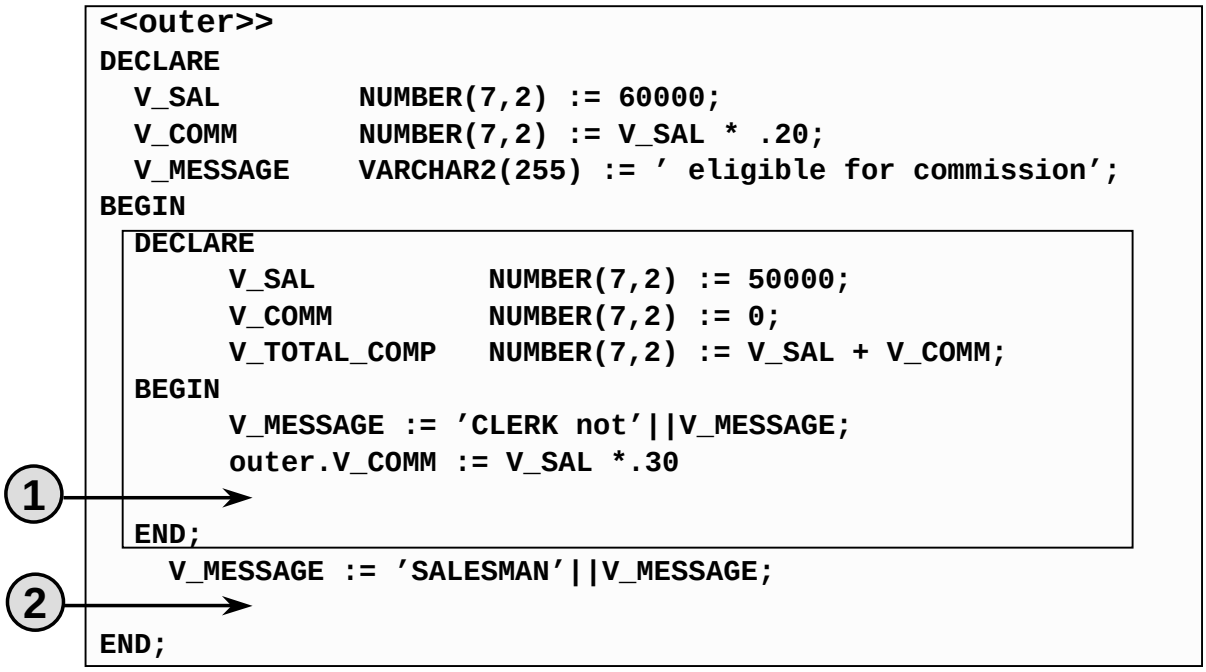
Qualify an Identifier

Qualify an identifier by using the block label prefix. In the example on the slide, the outer block is labeled `outer.` In the inner block, a variable with the same name, `birthdate,` as the variable in the outer block is declared. To reference the variable, `birthdate`, from the outer block in the inner block, prefix the variable by the block name, `outer.birthdate`.

For more information on block labels, see *PL/SQL User's Guide and Reference*, "Fundamentals."

Determining Variable Scope

Class Exercise



Class Exercise

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

1. The value of `V_MESSAGE` at position 1.
2. The value of `V_TOTAL_COMP` at position 2.
3. The value of `V_COMM` at position 1.
4. The value of `outer.V_COMM` at position 1.
5. The value of `V_COMM` at position 2.
6. The value of `V_MESSAGE` at position 2.

Operators in PL/SQL

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

} Same as in SQL

- Exponential operator (**)

ORACLE

Order of Operations

The operations within 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

Note: It is not necessary to use parentheses with Boolean expressions, but it does make the text easier to read. For more information on operators, see *PL/SQL User's Guide and Reference*, "Fundamentals."

Operators in PL/SQL

Examples:

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

```
v_count      := v_count + 1;
```

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

```
v_equal      := (v_n1 = v_n2);
```

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

```
v_valid      := (v_empno IS NOT NULL);
```

ORACLE

Operators in PL/SQL

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**

ORACLE

Programming Guidelines

Follow programming guidelines shown on 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 to help you distinguish keywords from named objects.

Category	Case Convention	Examples
SQL statements	Uppercase	`SELECT, INSERT`
PL/SQL keywords	Uppercase	`DECLARE`, `BEGIN`, `IF`
Datatypes	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
  v_deptno   NUMBER(4);
  v_location_id  NUMBER(4);
BEGIN
  SELECT  department_id,
          location_id
  INTO    v_deptno,
          v_location_id
  FROM    departments
  WHERE   department_name
          = 'Sales';
...
END;
/
```

ORACLE

Indenting Code

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

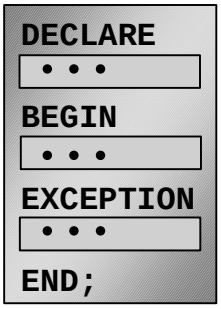
```
IF x>y THEN v_max:=x;ELSE v_max:=y;END IF;
```

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

Summary

In this lesson you should have learned the following:

- **PL/SQL block syntax and guidelines**
- **How to use identifiers correctly**
- **PL/SQL block structure: nesting blocks and scoping rules**
- **PL/SQL programming:**
  - **Functions**
  - **Data type conversions**
  - **Operators**
  - **Conventions and guidelines**



ORACLE

Summary

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

Identifiers are used to name PL/SQL program items and units, which include constants, variables, exceptions, cursors, cursor variables, subprograms, and packages.

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.

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. Generally, the form of the function follows the *datatype* `TO` *datatype* convention. The first data type is the input data type. The second data type is the output data type.

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 allow you to compare arbitrarily complex expressions.

Variables declared in *i*SQL*Plus are called bind variables. To reference these variables in PL/SQL programs, they should be preceded by a colon.

Practice 2 Overview

This practice covers the following topics:

- **Reviewing scoping and nesting rules**
- **Developing and testing PL/SQL blocks**

ORACLE

Practice 2 Overview

This practice reinforces the basics of PL/SQL that were presented in the lesson. The practices use sample PL/SQL blocks and test the understanding of the rules of scoping. Students also write and test PL/SQL blocks.

Paper-Based Questions

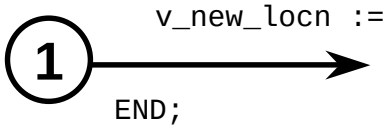
Questions 1 and 2 are paper-based questions.

Practice 2

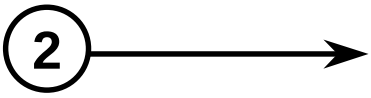
PL/SQL Block

```
DECLARE
  v_weight  NUMBER(3) := 600;
  v_message VARCHAR2(255) := 'Product 10012';
BEGIN

    DECLARE
      v_weight              NUMBER(3) := 1;
      v_message             VARCHAR2(255) := 'Product 11001';
      v_new_locn            VARCHAR2(50) := 'Europe';
    BEGIN
      v_weight := v_weight + 1;
      v_new_locn := 'Western ' || v_new_locn;
```




```
    END;
  v_weight := v_weight + 1;
  v_message := v_message || ' is in stock';
  v_new_locn := 'Western ' || v_new_locn;
```




```
    END;
```

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

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

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

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

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

Scope Example

```
DECLARE
  v_customer       VARCHAR2(50) := 'Womansport';
  v_credit_rating  VARCHAR2(50) := 'EXCELLENT';
BEGIN
  DECLARE
    v_customer NUMBER(7) := 201;
    v_name     VARCHAR2(25) := 'Unisports';
  BEGIN
    v_customer      v_name    v_credit_rating
    END;
     v_customer     v_name    v_credit_rating
END;
```

2. Suppose you embed a subblock within a block, as shown above. You declare two variables, V_CUSTOMER and V_CREDIT_RATING, in the main block. You also declare two variables, V_CUSTOMER and V_NAME, in the subblock. Determine the values and data types for each of the following cases.
   a. The value of V_CUSTOMER in the subblock is:

   b. The value of V_NAME in the subblock is:

   c. The value of V_CREDIT_RATING in the subblock is:

   d. The value of V_CUSTOMER in the main block is:

   e. The value of V_NAME in the main block is:

   f. The value of V_CREDIT_RATING in the main block is:

Practice 2 (continued)

3. Create and execute a PL/SQL block that accepts two numbers through *i*SQL*Plus substitution variables. Use the `DEFINE` command to provide the two values. Pass these two values to the PL/SQL block through *i*SQL*Plus substitution variables. The first number should be divided by the second number and have the second number added to the result. The result should be stored in a PL/SQL variable and printed on the screen.

   Note: `SET VERIFY OFF` in the PL/SQL block.

```
DEFINE p_num1 = 2
DEFINE p_num2 = 4
```

```
4.5
PL/SQL procedure successfully completed.
```

4. Build a PL/SQL block that computes the total compensation for one year. The annual salary and the annual bonus percentage values are defined using the `DEFINE` command and are passed to the PL/SQL block through *i*SQL*Plus substitution variables. The bonus must be converted from a whole number to a decimal (for example, 15 to .15). If the salary is `null`, set it to zero before computing the total compensation. Execute the PL/SQL block. *Reminder:* Use the `NVL` function to handle `null` values.

   Note: To test the `NVL` function, set the `DEFINE` variable equal to `NULL`.

```
DEFINE p_salary=50000
DEFINE p_bonus=10
```

```
55000
PL/SQL procedure successfully completed.
```

3

Interacting with the Oracle Server

ORACLE

Objectives

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

- **Write a successful SELECT statement in PL/SQL**
- **Write DML statements in PL/SQL**
- **Control transactions in PL/SQL**
- **Determine the outcome of SQL Data Manipulation Language (DML) statements**

ORACLE

Lesson Aim

In this lesson, you learn to embed standard SQL `SELECT`, `INSERT`, `UPDATE`, and `DELETE` statements in PL/SQL blocks. You also learn to control transactions and determine the outcome of SQL data manipulation language (DML) statements in PL/SQL.

SQL Statements in PL/SQL

- **Extract a row of data 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.**
- **Determine DML outcome with implicit cursor attributes.**

ORACLE

SQL Statements in PL/SQL

When you extract information from or apply changes to the database, you must use SQL. PL/SQL supports data manipulation language and transaction control commands of SQL. You can use SELECT statements to populate variables with values queried from a row in a table. You can use DML commands to modify the data in a database table. However, remember the following points about PL/SQL blocks 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 does not support data control language (DCL) statements, such as GRANT or REVOKE.

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];
```

ORACLE

Retrieving Data Using PL/SQL

Use the SELECT statement to retrieve data from the database. In the syntax:

select_list	is a list of at least one column and can include SQL expressions, row functions, or group functions.
variable_name	is the scalar variable that holds the retrieved value.
record_name	is the 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 (;).
- The INTO clause is required for the SELECT statement when it is embedded in PL/SQL.
- The WHERE clause is optional and can be used to specify input variables, constants, literals, or PL/SQL expressions.

Retrieving Data Using PL/SQL (continued)

- Specify the same number of variables in the `INTO` clause as 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 one and only one row.

Example:

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

ORACLE

SELECT Statements in PL/SQL

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 One and 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 one and only one row. A query that returns more than one row or no row generates an error.

PL/SQL manages with these errors by raising standard exceptions, which you can trap in the exception section of the block with the `NO_DATA_FOUND` and `TOO_MANY_ROWS` exceptions (exception handling is covered in a subsequent lesson). Code `SELECT` statements to return a single row.

Retrieving Data in PL/SQL

Retrieve the hire date and the salary for the specified employee.

Example:

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

ORACLE

Retrieving Data in PL/SQL

In the example on the slide, the variables `v_hire_date` and `v_salary` are declared in the `DECLARE` 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 is retrieved from the `EMPLOYEES` table and stored in the `v_hire_date` and `v_salary` variables, respectively. Observe how the `INTO` clause, along with the `SELECT` statement, retrieves the database column values into the PL/SQL variables.

Retrieving Data in PL/SQL

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

Example:

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

ORACLE

Retrieving Data in PL/SQL

In the example on the slide, the `v_sum_sal` and `v_deptno` variables are declared in the `DECLARE` section of the PL/SQL block. In the executable section, the total salary for the department with the `DEPARTMENT_ID` 60 is computed using the SQL aggregate function `SUM`, and assigned to the `v_sum_sal` variable. Note that group functions cannot be used in PL/SQL syntax. They are used in SQL statements within a PL/SQL block.

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

```
The sum 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
```

ORACLE

Naming Conventions

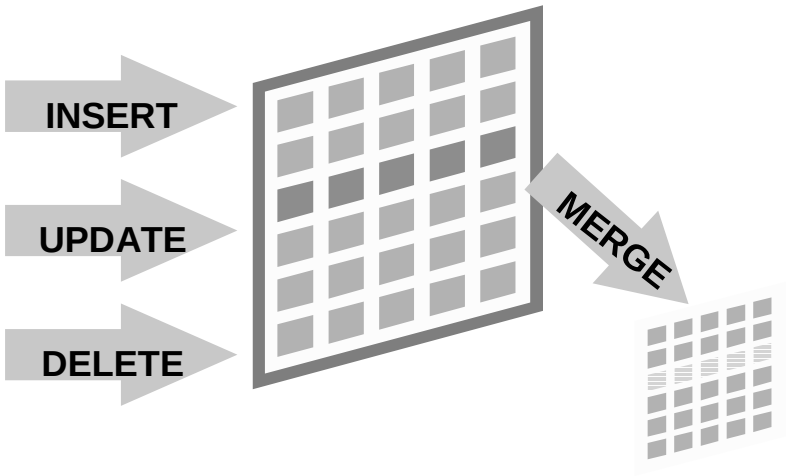
In potentially ambiguous SQL statements, the names of database columns take precedence over the names of local variables. The example shown on the slide is defined as follows: Retrieve the hire date and today's date from the EMPLOYEES table for employee ID 176. This example raises an unhandled run-time exception because in the WHERE clause, the PL/SQL variable names are the same as that of the database column names in the EMPLOYEES table.

The following DELETE statement removes all employees from the EMPLOYEES table where last name is not null, not just 'King', because the Oracle server assumes that both LAST_NAMES in the WHERE clause refer to the database column:

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

Manipulating Data Using PL/SQL

- **Make changes to database tables by using DML commands:**
  - **INSERT**
  - **UPDATE**
  - **DELETE**
  - **MERGE**



ORACLE

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 of data to the table.
- The `UPDATE` statement modifies existing rows in the table.
- The `DELETE` statement removes unwanted 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 on 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;
/
```

ORACLE

Inserting Data

In the example on 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 database sequences
- Derive values in the PL/SQL block
- Add column default values

Note: There is no possibility for ambiguity with identifiers and column names in the INSERT statement. Any identifier in the INSERT clause must be a database column name.

Updating Data

Increase the salary of all employees who are stock clerks.

Example:

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

END;
/
```

ORACLE

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.

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 =. 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.

Deleting Data

Delete rows that belong to department 10 from the EMPLOYEES table.

Example:

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

ORACLE

Deleting Data

The DELETE statement removes unwanted rows from a table. Without the use of a WHERE clause, the entire contents of a table can be removed, provided there are no integrity constraints.

Merging Rows

Insert or update rows in the COPY_EMP table to match the EMPLOYEES table.

```
DECLARE
      v_empno EMPLOYEES.EMPLOYEE_ID%TYPE := 100;
BEGIN
MERGE INTO copy_emp c
     USING employees e
     ON (c.employee_id = v_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;
```

ORACLE

Merging Rows

The MERGE statement inserts or updates rows in one table, using data from another table. Each row is inserted or updated in the target table, depending upon 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 in the next page.

Merging Data (continued)

```
DECLARE
       v_empno EMPLOYEES.EMPLOYEE_ID%TYPE := 100;
BEGIN
MERGE INTO copy_emp c
     USING employees e
     ON (c.employee_id = v_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;
/
```

Naming Conventions

- **Use a naming convention to avoid ambiguity in the `WHERE` clause.**
- **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.**
- **The names of local variables and formal parameters take precedence over the names of database tables.**
- **The names of columns take precedence over the names of local variables.**

Naming Conventions

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.

Naming Conventions (continued)

The following table shows a set of prefixes and suffixes that distinguish identifiers from other identifiers, database objects, and from other named objects.

Identifier	Naming Convention	Example
Variable	`v_name`	`v_sal`
Constant	`c_name`	`c_company_name`
Cursor	`name_cursor`	`emp_cursor`
Exception	`e_name`	`e_too_many`
Table type	`name_table_type`	`amount_table_type`
Table	`name_table`	`countries`
Record type	`name_record_type`	`emp_record_type`
Record	`name_record`	`customer_record`
*i*SQL*Plus substitution variable (also referred to as substitution parameter)	`p_name`	`p_sal`
*i*SQL*Plus host or bind variable	`g_name`	`g_year_sal`

In such cases, to avoid ambiguity, prefix the names of local variables and formal parameters with `v_`, as follows:

```
DECLARE
   v_last_name VARCHAR2(25);
```

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.

SQL Cursor

- A cursor is a private SQL work area.
- There are two types of cursors:
  - Implicit cursors
  - Explicit cursors
- The Oracle server uses implicit cursors to parse and execute your SQL statements.
- Explicit cursors are explicitly declared by the programmer.

ORACLE

SQL Cursor

Whenever you issue a SQL statement, the Oracle server opens an area of memory in which the command is parsed and executed. This area is called a cursor.

When the executable part of a block issues a SQL statement, PL/SQL creates an implicit cursor, which PL/SQL manages automatically. The programmer explicitly declares and names an explicit cursor. There are four attributes available in PL/SQL that can be applied to cursors.

Note: More information about explicit cursors is covered in a subsequent lesson.

For more information, refer to *PL/SQL User's Guide and Reference*, "Interaction with Oracle."

SQL Cursor Attributes

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

SQL%ROWCOUNT	**Number of rows affected by the most recent SQL statement (an integer value)**
SQL%FOUND	**Boolean attribute that evaluates to TRUE if the most recent SQL statement affects one or more rows**
SQL%NOTFOUND	**Boolean attribute that evaluates to TRUE if the most recent SQL statement does not affect any rows**
SQL%ISOPEN	**Always evaluates to FALSE because PL/SQL closes implicit cursors immediately after they are executed**

SQL Cursor Attributes

SQL cursor attributes allow 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 use the attributes `SQL%ROWCOUNT`, `SQL%FOUND`, `SQL%NOTFOUND`, and `SQL%ISOPEN` in the exception section of a block to gather information about the execution of a DML statement. PL/SQL does not return an error if a DML statement does not affect any rows in the underlying table. However, if a `SELECT` statement does not retrieve any rows PL/SQL returns an exception.

SQL Cursor Attributes

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

ORACLE

SQL Cursor Attributes (continued)

The example on the slide deletes the rows from the EMPLOYEES table for EMPLOYEE_ID 176. Using the SQL%ROWCOUNT attribute, you can print the number of rows deleted.

Transaction Control Statements

- **Initiate a transaction with the first DML command to follow a `COMMIT` or `ROLLBACK`.**
- **Use `COMMIT` and `ROLLBACK` SQL statements to terminate a transaction explicitly.**

ORACLE

Transaction Control Statements

You control the logic of transactions with `COMMIT` and `ROLLBACK` SQL statements, rendering some groups of database changes permanent while discarding others. As with Oracle server, DML transactions start at the first command that follows a `COMMIT` or `ROLLBACK`, and end on the next successful `COMMIT` or `ROLLBACK`. These actions may occur within a PL/SQL block or as a result of events in the host environment (for example, in most cases, ending a *i*SQL*Plus session automatically commits the pending transaction). To mark an intermediate point in the transaction processing, use `SAVEPOINT`.

Syntax

```
COMMIT [WORK];
SAVEPOINT savepoint_name;
ROLLBACK [WORK];
ROLLBACK [WORK] TO [SAVEPOINT] savepoint_name;
```

where: `WORK` is for compliance with ANSI standards.

Note: The transaction control commands are all valid within PL/SQL, although the host environment may place some restriction on their use.

You can also include explicit locking commands (such as `LOCK TABLE` and `SELECT ... FOR UPDATE`) in a block, which stays in effect until the end of the transaction (a subsequent lesson covers more information on the `FOR UPDATE` command). Also, one PL/SQL block does not necessarily imply one transaction.

Summary

In this lesson you should have learned the following:

- **Embed SQL in the PL/SQL block using `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `MERGE`**
- **Embed transaction control statements in a PL/SQL block `COMMIT`, `ROLLBACK`, `SAVEPOINT`**

ORACLE

Summary

The DML commands `INSERT`, `UPDATE`, `DELETE`, and `MERGE` can be used in PL/SQL programs without any restriction. The `COMMIT` statement ends the current transaction and makes permanent any changes made during that transaction. The `ROLLBACK` statement ends the current transaction and cancels any changes that were made during that transaction. `SAVEPOINT` names and marks the current point in the processing of a transaction. With the `ROLLBACK TO SAVEPOINT` statement, you can undo parts of a transaction instead of the whole transaction.

Summary

In this lesson you should have learned the following:

- **There are two cursor types: implicit and explicit.**
- **Implicit cursor attributes are used to verify the outcome of DML statements:**
  - **SQL%ROWCOUNT**
  - **SQL%FOUND**
  - **SQL%NOTFOUND**
  - **SQL%ISOPEN**
- **Explicit cursors are defined by the programmer.**

ORACLE

Summary (continued)

An implicit cursor is declared by PL/SQL for each SQL data manipulation statement. Every implicit cursor has four attributes: `%FOUND`, `%ISOPEN`, `%NOTFOUND`, and `%ROWCOUNT`. When appended to the cursor or cursor variable, these attributes return useful information about the execution of a DML statement. You can use cursor attributes in procedural statements but not in SQL statements. Explicit cursors are defined by the programmer.

Practice 3 Overview

This practice covers the following topics:

- **Creating a PL/SQL block to select data from a table**
- **Creating a PL/SQL block to insert data into a table**
- **Creating a PL/SQL block to update data in a table**
- **Creating a PL/SQL block to delete a record from a table**

ORACLE

Practice 3 Overview

In this practice you write PL/SQL blocks to select, input, update, and delete information in a table, using basic SQL query and DML statements within a PL/SQL block.

Practice 3

1. Create a PL/SQL block that selects the maximum department number in the `DEPARTMENTS` table and stores it in an *i*SQL*Plus variable. Print the results to the screen. Save your PL/SQL block in a file named `p3q1.sql`. by clicking the `Save Script` button. Save the script with a `.sql` extension.

G_MAX_DEPTNO
270

2. Modify the PL/SQL block you created in exercise 1 to insert a new department into the `DEPARTMENTS` table. Save the PL/SQL block in a file named `p3q2.sql` by clicking the `Save Script` button. Save the script with a `.sql` extension.
   a. Rather than printing the department number retrieved from exercise 1, add 10 to it and use it as the department number for the new department.
   b. Use the `DEFINE` command to provide the department name. Name the new department `Education`. Pass the value to the PL/SQL block through a *i*SQL*Plus substitution variable.
   c. Leave the location number as null for now.
   d. Execute the PL/SQL block.
   e. Display the new department that you created.

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
280	Education		

3. Create a PL/SQL block that updates the location ID for the new department that you added in the previous practice. Save your PL/SQL block in a file named `p3q3.sql` by clicking the `Save Script` button. Save the script with a `.sql` extension.
   a. Use an *i*SQL*Plus variable for the department ID number that you added in the previous practice.
   b. Use the `DEFINE` command to provide the location ID. Name the new location id 1700. Pass the value to the PL/SQL block through a *i*SQL*Plus substitution variable.
   c. Test the PL/SQL block.

```
DEFINE p_deptno = 280
DEFINE p_loc = 1700
```

   d. Display the department number, department name, and location for the updated department.

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
280	Education		1700

Practice 3 (continued)

4. Create a PL/SQL block that deletes the department that you created in exercise 2. Save the PL/SQL block in a file named `p3q4.sql`. by clicking the `Save Script` button. Save the script with a `.sql` extension.
   a. Use the `DEFINE` command to provide the department ID. Pass the value to the PL/SQL block through a *i*SQL*Plus substitution variable.
   b. Print to the screen the number of rows affected.
   c. Test the PL/SQL block.

```
DEFINE p_deptno=280
```

G_RESULT
1 row(s) deleted.

   d. Confirm that the department has been deleted.

no rows selected

4 Writing Control Structures

ORACLE

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` expressions**
- **Construct and identify different loop statements**
- **Use logic tables**
- **Control block flow using nested loops and labels**

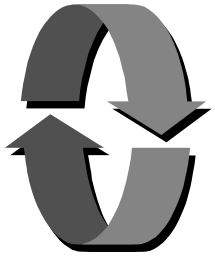
ORACLE

Lesson Aim

In this lesson, you learn about conditional control within the PL/SQL block by using `IF` statements and loops.

Controlling PL/SQL Flow of Execution

- **You can change the logical execution of statements using conditional `IF` statements and loop control structures.**
- **Conditional `IF` statements:**
  - **`IF-THEN-END IF`**
  - **`IF-THEN-ELSE-END IF`**
  - **`IF-THEN-ELSIF-END IF`**



ORACLE

Controlling PL/SQL 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 (covered later in this lesson).

There are three forms of `IF` statements:

- `IF-THEN-END IF`
- `IF-THEN-ELSE-END IF`
- `IF-THEN-ELSIF-END IF`

IF Statements

Syntax:

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

If the employee name is Gietz, set the Manager ID to 102.

```
IF UPPER(v_last_name) = 'GIETZ' THEN
  v_mgr := 102;
END IF;
```

ORACLE

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 (TRUE, FALSE, or NULL). (It is associated with a sequence of statements, which is executed only if the expression yields TRUE.)
THEN	is a clause that associates the Boolean expression that precedes it 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.)
ELSIF	is a keyword that introduces a Boolean expression. (If the first condition yields FALSE or NULL then the ELSIF keyword introduces additional conditions.)
ELSE	is a keyword that executes the sequence of statements that follows it if the control reaches it.

Simple IF Statements

If the last name is Vargas:

- **Set job ID to SA_REP**
- **Set department number to 80**

```
. . .
IF v_ename      = 'Vargas' THEN
   v_job       := 'SA_REP';
   v_deptno    := 80;
END IF;
. . .
```

ORACLE

Simple IF Statements

In the example on the slide, PL/SQL assigns values to the following variables, only if the condition is TRUE:

- v_job and v_deptno

If the condition is FALSE or NULL, PL/SQL ignores the statements in the IF block. In either case, control resumes at the next statement in the program following the END IF.

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.

Compound IF Statements

If the last name is Vargas and the salary is more than 6500:
Set department number to 60

```
. . .
IF v_ename      = 'Vargas' AND salary > 6500 THEN
   v_deptno    := 60;
END IF;
. . .
```

ORACLE

Compound IF Statements

Compound IF statements use logical operators like AND and NOT. In the example on the slide, the IF statement has two conditions to evaluate:

- Last name should be Vargas
- Salary should be greater than 6500

Only if both the above conditions are evaluated as TRUE, v_deptno is set to 60.

Consider the following example:

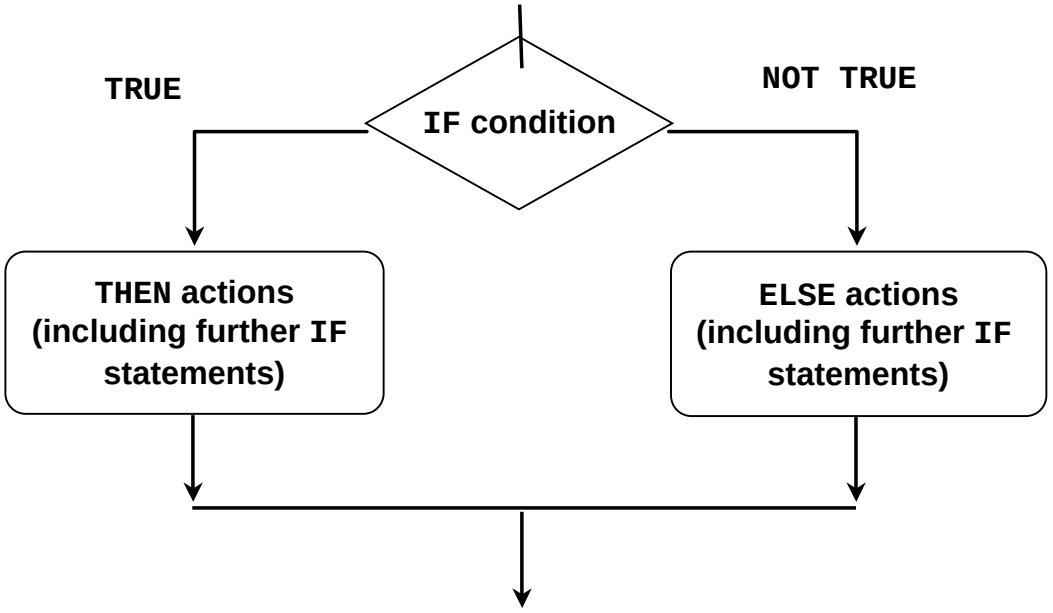
```
. . .
IF v_department = '60' OR v_hiredate > '01-Dec-1999' THEN
      v_mgr := 101;
END IF;
. . .
```

In the above example , the IF statement has two conditions to evaluate:

- Department ID should be 60
- Hire date should be greater than 01-Dec-1999

If either of the above conditions are evaluated as TRUE, v_mgr is set to 101.

IF-THEN-ELSE Statement Execution Flow



ORACLE

IF-THEN-ELSE Statement Execution Flow

While writing an IF construct, if the condition is FALSE or NULL, you can use the ELSE clause to carry out other actions. As with the simple IF statement, control resumes in the program from the END IF clause. For example:

```
IF condition1 THEN
  statement1;
ELSE
  statement2;
END IF;
```

Nested IF Statements

Either set of actions of the result of the first IF statement can include further IF statements before specific actions are performed. The THEN and ELSE clauses can include IF statements. Each nested IF statement must be terminated with a corresponding END IF clause.

```
IF condition1 THEN
  statement1;
ELSE
  IF condition2 THEN
    statement2;
  END IF;
END IF;
```

IF-THEN-ELSE Statements

Set a Boolean flag to TRUE if the hire date is greater than five years; otherwise, set the Boolean flag to FALSE.

```
DECLARE
        v_hire_date  DATE := '12-Dec-1990';
        v_five_years BOOLEAN;
BEGIN
. . .
IF MONTHS_BETWEEN(SYSDATE,v_hire_date)/12 > 5 THEN
   v_five_years := TRUE;
ELSE
   v_five_years := FALSE;
END IF;
...
```

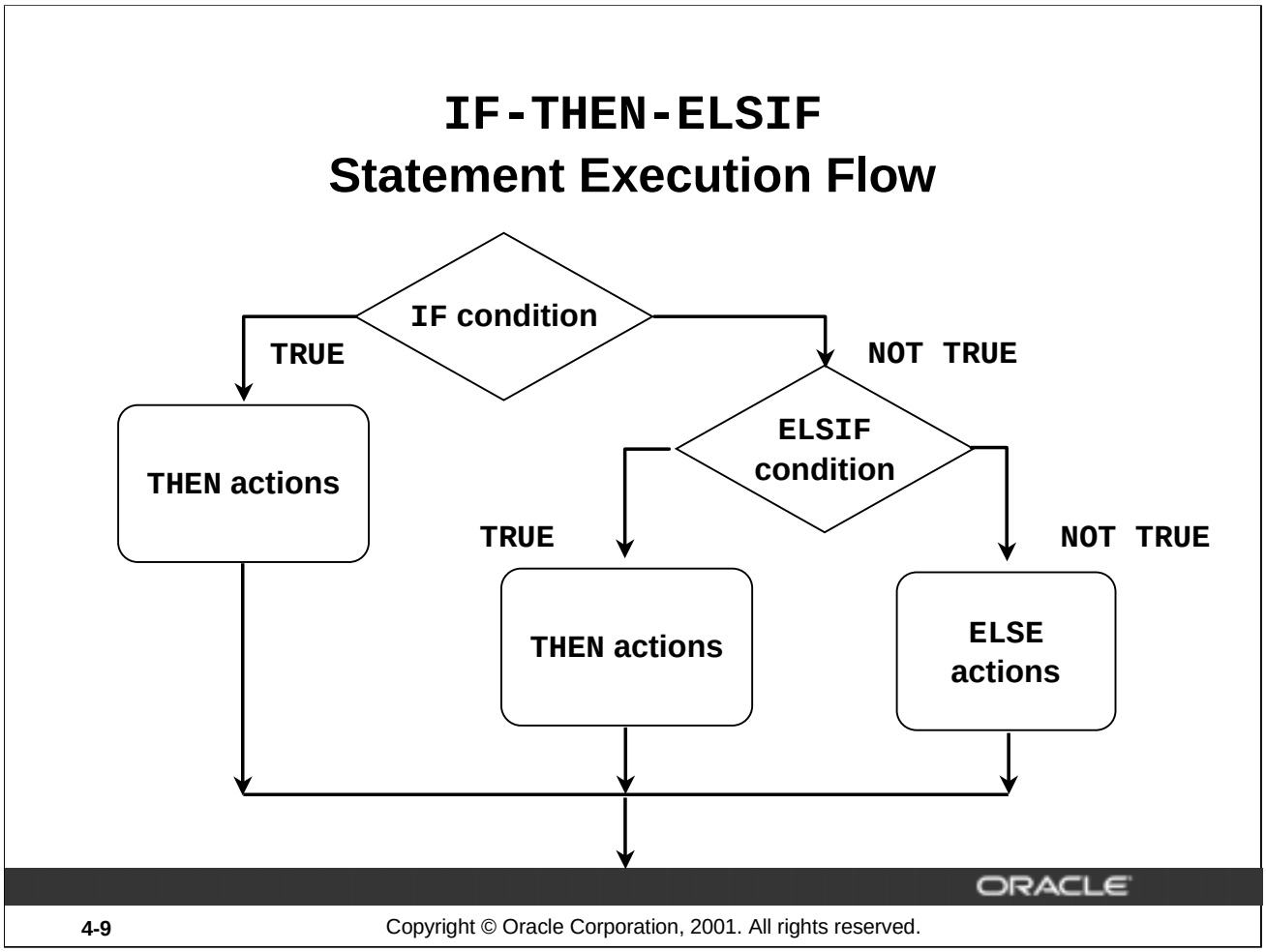
ORACLE

IF-THEN-ELSE Statements: Example

In the example on the slide, the MONTHS_BETWEEN function is used to find out the difference in months between the current date and the v_hire_date variable. Because the result is the difference of the number of months between the two dates, the resulting value is divided by 12 to convert the result into years. If the resulting value is greater than 5, the Boolean flag is set to TRUE; otherwise, the Boolean flag is set to FALSE.

Consider the following example: Check the value in the v_ename variable. If the value is King, set the v_job variable to AD_PRES. Otherwise, set the v_job variable to ST_CLERK.

```
IF v_ename  = 'King' THEN
   v_job   := 'AD_PRES';
ELSE
   v_job   := 'ST_CLERK';
END IF;
```



IF-THEN-ELSIF Statement Execution Flow

Sometimes you want to select an action from several mutually exclusive alternatives. The third form of IF statement uses the keyword ELSIF (not ELSEIF) to introduce additional conditions, as follows:

```
IF condition1 THEN
   sequence_of_statements1;
ELSIF condition2 THEN
   sequence_of_statements2;
ELSE
   sequence_of_statements3;
END IF;
```

IF-THEN-ELSIF Statement Execution Flow (continued)

If the first condition is false or null, the ELSIF clause tests another condition. An IF statement can have any number of ELSIF clauses; the final ELSE clause is optional. Conditions are evaluated one by one from top to bottom. If any condition is true, its associated sequence of statements is executed and control passes to the next statement. If all conditions are false or null, the sequence in the ELSE clause is executed. Consider the following example: Determine an employee's bonus based upon the employee's department.

```
IF v_deptno = 10 THEN
  v_bonus := 5000;
ELSIF v_deptno = 80 THEN
  v_bonus := 7500;
ELSE
  v_bonus := 2000;
END IF;
```

Note: In case of multiple IF – ELSIF statements only the first true statement is processed.

IF-THEN-ELSIF Statements

For a given value, calculate a percentage of that value based on a condition.

Example:

```
. . .
IF    v_start > 100 THEN
      v_start := .2 * v_start;
ELSIF v_start >= 50 THEN
      v_start := .5 * v_start;
ELSE
      v_start := .1 * v_start;
END IF;
. . .
```

ORACLE

IF-THEN-ELSIF Statements

When possible, use the ELSIF clause instead of nesting IF statements. The code is easier to read and understand, and the logic is clearly identified. If the action in the ELSE clause consists purely of another IF statement, it is more convenient to use the ELSIF clause. This makes the code clearer by removing the need for nested END IF statements at the end of each further set of conditions and actions.

Example

```
IF condition1 THEN
  statement1;
ELSIF condition2 THEN
  statement2;
ELSIF condition3 THEN
  statement3;
END IF;
```

The example IF-THEN-ELSIF statement above is further defined as follows:

For a given value, calculate a percentage of the original value. If the value is more than 100, then the calculated value is two times the starting value. If the value is between 50 and 100, then the calculated value is 50% of the starting value. If the entered value is less than 50, then the calculated value is 10% of the starting value.

Note: Any arithmetic expression containing null values evaluates to null.

CASE Expressions

- **A CASE expression selects a result and returns it.**
- **To select the result, the CASE expression uses an expression whose value 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;
```

ORACLE

CASE Expressions

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

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. Also, its WHEN clauses contain search conditions that yield a Boolean value, not expressions that can yield a value of any type.

CASE Expressions: Example

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

ORACLE

CASE Expressions: Example

In the example on the slide, the CASE expression uses the value in the `v_grade` variable as the expression. This value is accepted from the user using a substitution variable. Based on the value entered by the user, the CASE expression evaluates the value of the `v_appraisal` variable based on the value of the `v_grade` value. The output of the above example will be as follows:

```
old 2: v_grade CHAR(1) := UPPER('&p_grade');
new 2: v_grade CHAR(1) := UPPER('a');
Grade: A Appraisal Excellent
PL/SQL procedure successfully completed.
```

CASE Expressions: Example (continued)

If the example on the slide is written using a searched CASE expression it will look like this:

```
DEFINE p_grade = a
DECLARE
   v_grade CHAR(1) := UPPER('&p_grade');
   v_appraisal VARCHAR2(20);
BEGIN
    v_appraisal :=
      CASE
         WHEN v_grade = 'A' THEN 'Excellent'
         WHEN v_grade = 'B' THEN 'Very Good'
         WHEN v_grade = 'C' THEN 'Good'
         ELSE 'No such grade'
     END;
DBMS_OUTPUT.PUT_LINE
('Grade: '|| v_grade || ' Appraisal ' || v_appraisal);
END;
/
```

Handling Nulls

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

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

ORACLE

Handling Nulls

In the following example, you might 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.

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

In the next example, you might expect the sequence of statements to execute because *a* and *b* seem equal. But, again, that is unknown, so 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;  -- not executed
END IF;
```

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

ORACLE

Boolean Conditions with Logical Operators

You can build a simple Boolean condition by combining number, character, or 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. 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 evaluate 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 V_FLAG in each case?

```
v_flag := v_reorder_flag AND v_available_flag;
```

V_REORDER_FLAG	V_AVAILABLE_FLAG	V_FLAG
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
NULL	TRUE	NULL
NULL	FALSE	FALSE

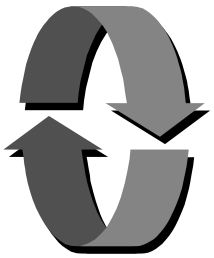
ORACLE

Building Logical Conditions

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

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**



ORACLE

Iterative Control: LOOP Statements

PL/SQL provides a number of facilities to structure loops to repeat a statement or sequence of statements multiple times.

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

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

Use the EXIT statement to terminate loops.

For more information, refer to *PL/SQL User's Guide and Reference*, "Control Structures."

Note: Another type of FOR LOOP, cursor FOR LOOP, is discussed in a subsequent lesson.

Basic Loops

Syntax:

```
LOOP                                    -- delimiter
  statement1;                           -- statements
  . . .
  EXIT [WHEN condition];                -- EXIT statement
END LOOP;                               -- delimiter
```

```
condition    is a Boolean variable or
             expression (TRUE, FALSE,
                         or NULL);
```

ORACLE

Basic Loops

The simplest form of 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 statement at least once, even if the condition is already met upon entering the loop. Without the EXIT statement, the loop would be infinite.

The 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 allow 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.

Basic Loops

Example:

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

ORACLE

Basic Loops (continued)

The basic loop example shown on the slide is defined as follows: Insert three new locations IDs for the country code of CA 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, provided 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 that condition is true, the loop will exit and the statements will never execute.

WHILE Loops

Syntax:

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

Condition is evaluated at the beginning of each iteration.

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

ORACLE

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. If the condition is FALSE at the start of the loop, then 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, then 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
  v_country_id      locations.country_id%TYPE := 'CA';
  v_location_id     locations.location_id%TYPE;
  v_city            locations.city%TYPE := 'Montreal';
  v_counter         NUMBER  := 1;
BEGIN
  SELECT MAX(location_id) INTO v_location_id FROM locations
  WHERE country_id = v_country_id;
  WHILE v_counter <= 3 LOOP
    INSERT INTO locations(location_id, city, country_id)
    VALUES((v_location_id + v_counter), v_city, v_country_id );
    v_counter := v_counter + 1;
  END LOOP;
END;
/
```

ORACLE

WHILE Loops (continued)

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

With each iteration through the WHILE loop, a counter (v_counter) is incremented. If the number of iterations is less than or equal to the number 3, the code within the loop is executed and a row is inserted into the LOCATIONS table. After the counter exceeds the number of items for this location, the condition that controls the loop evaluates to FALSE and the loop is terminated.

FOR Loops

Syntax:

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

- **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.**

ORACLE

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 determine 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 that 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.

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 must evaluate to integers. 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 will not be executed, provided REVERSE has not been used.

For example the following, statement is executed only once:

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

FOR Loops

Insert three new locations IDs for the country code of CA and the city of Montreal.

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

ORACLE

FOR Loops (continued)

The example shown on the slide is defined as follows: Insert three new locations for the country code of CA and the city of Montreal.

This is done using a 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.**

ORACLE

FOR Loops (continued)

The slide lists the guidelines to follow while writing a FOR Loop.

Note: While writing a FOR loop, 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
  v_lower   NUMBER := 1;
  v_upper   NUMBER := 100;
BEGIN
  FOR i IN v_lower..v_upper LOOP
  ...
  END LOOP;
END;
```

Guidelines While Using Loops

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

ORACLE

Guidelines While Using 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, then 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

- **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.**

ORACLE

Nested Loops and Labels

You can nest loops to multiple levels. 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. Label loops by placing the label before the word `LOOP` within label delimiters (<<*label*>>).

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
    v_counter := v_counter+1;
  EXIT WHEN v_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;
```

ORACLE

Nested Loops and Labels (continued)

In the example on 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` statement for clarity.

Summary

In this lesson you should have learned how to do the following:

Change the logical flow of statements by using control structures.

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

ORACLE

Summary

A conditional control construct checks for the validity of a condition and performs a corresponding 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 4 Overview

This practice covers the following topics:

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

ORACLE

Practice 4 Overview

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

Practice 4

1. Execute the command in the file `lab04_1.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. Select from the `MESSAGES` table to verify that your PL/SQL block worked.

RESULTS
1
2
3
4
5
7
9
10

8 rows selected.

2. Create a PL/SQL block that computes the commission amount for a given employee based on the employee's salary.
   a. Use the `DEFINE` command to provide the employee ID. Pass the value to the PL/SQL block through a *i*SQL*Plus substitution variable.

      ```
      DEFINE p_empno = 100
      ```

   b. If the employee's salary is less than $5,000, display the bonus amount for the employee as 10% of the salary.
   c. If the employee's salary is between $5,000 and $10,000, display the bonus amount for the employee as 15% of the salary.
   d. If the employee's salary exceeds $10,000, display the bonus amount for the employee as 20% of the salary.
   e. If the employee's salary is `NULL`, display the bonus amount for the employee as 0.
   f. Test the PL/SQL block for each case using the following test cases, and check each bonus amount.

Employee Number	Salary	Resulting Bonus
100	24000	4800
149	10500	2100
178	7000	1050

Note: Include `SET VERIFY OFF` in your solution.

If you have time, complete the following exercises:

3. Create an `EMP` table that is a replica of the `EMPLOYEES` table. You can do this by executing the script `lab4_3.sql`. Add a new column, `STARS`, of `VARCHAR2` data type and length of 50 to the `EMP` table for storing asterisk (*).

```
Table altered.
```

4. Create a PL/SQL block that rewards an employee by appending an asterisk in the `STARS` column for every $1000 of the employee's salary. Save your PL/SQL block in a file called `p4q4.sql` by clicking on the `Save Script` button. Remember to save the script with a `.sql` extension.
   a. Use the `DEFINE` command to provide the employee ID. Pass the value to the PL/SQL block through a *i*SQL*Plus substitution variable.
   b. Initialize a `v_asterisk` variable that contains a `NULL`.
   c. Append an asterisk to the string for every $1000 of the salary amount. For example, if the employee has a salary amount of $8000, the string of asterisks should contain eight asterisks. If the employee has a salary amount of $12500, the string of asterisks should contain 13 asterisks.
   d. Update the `STARS` column for the employee with the string of asterisks.
   e. Commit.
   f. Test the block for the following values:

```
DEFINE p_empno=104
DEFINE p_empno=174
DEFINE p_empno=176
```

   g. Display the rows from the `EMP` table to verify whether your PL/SQL block has executed successfully.

EMPLOYEE_ID	SALARY	STARS
104	6000	******
174	11000	**********
176	8600	*********

Note: `SET VERIFY OFF` in the PL/SQL block

5

Working with Composite Data Types

Objectives

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

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

ORACLE

Lesson Aim

In this lesson, you learn more about composite data types and their uses.

Composite Data Types

- **Are of two types:**
  - **PL/SQL RECORDs**
  - **PL/SQL Collections**
    - **INDEX BY Table**
    - **Nested Table**
    - **VARRAY**
- **Contain internal components**
- **Are reusable**

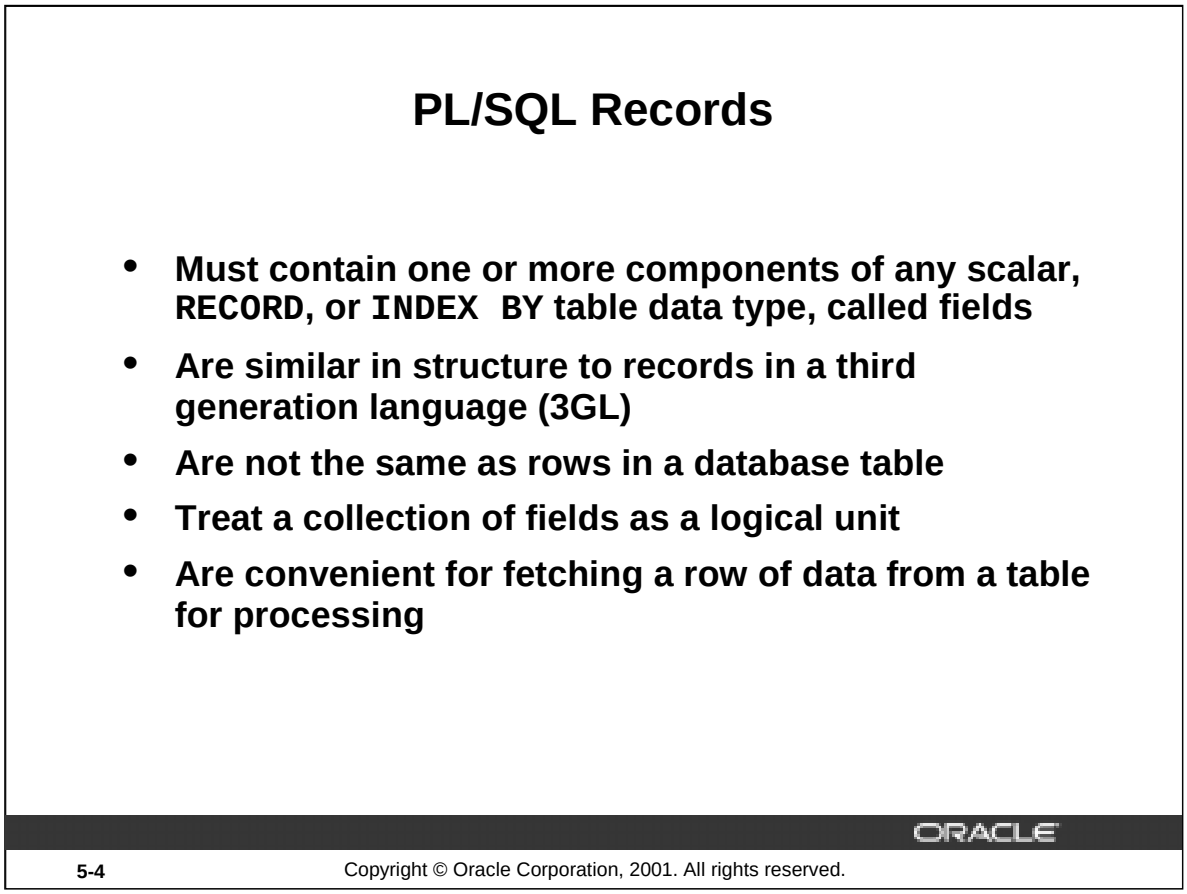
ORACLE

RECORD and TABLE Data Types

Like scalar variables, composite variables have a data type. Composite data types (also known as collections) are RECORD, TABLE, NESTED TABLE, and VARRAY. You use the RECORD data type to treat related but dissimilar data as a logical unit. You use the TABLE data type to reference and manipulate collections of data as a whole object. The NESTED TABLE and VARRAY data types are covered in the *Advanced PL/SQL* course.

A record is a group of related data items stored as fields, each with its own name and data type. A table contains a column and a primary key to give you array-like access to rows. After they are defined, tables and records can be reused.

For more information, refer to *PL/SQL User's Guide and Reference,* "Collections and Records."



PL/SQL Records

A *record* is a group of related data items stored in *fields*, each with its own name and data type. For example, suppose you have different kinds of data about an employee, such as name, salary, hire date, and so on. This data is dissimilar in type but logically related. A record that contains such fields as the name, salary, and hire date of an employee allows you to treat the data as a logical unit. When you declare a record type for these fields, they can be manipulated as a unit.

- 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:

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

Where *field_declaration* is:

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

ORACLE

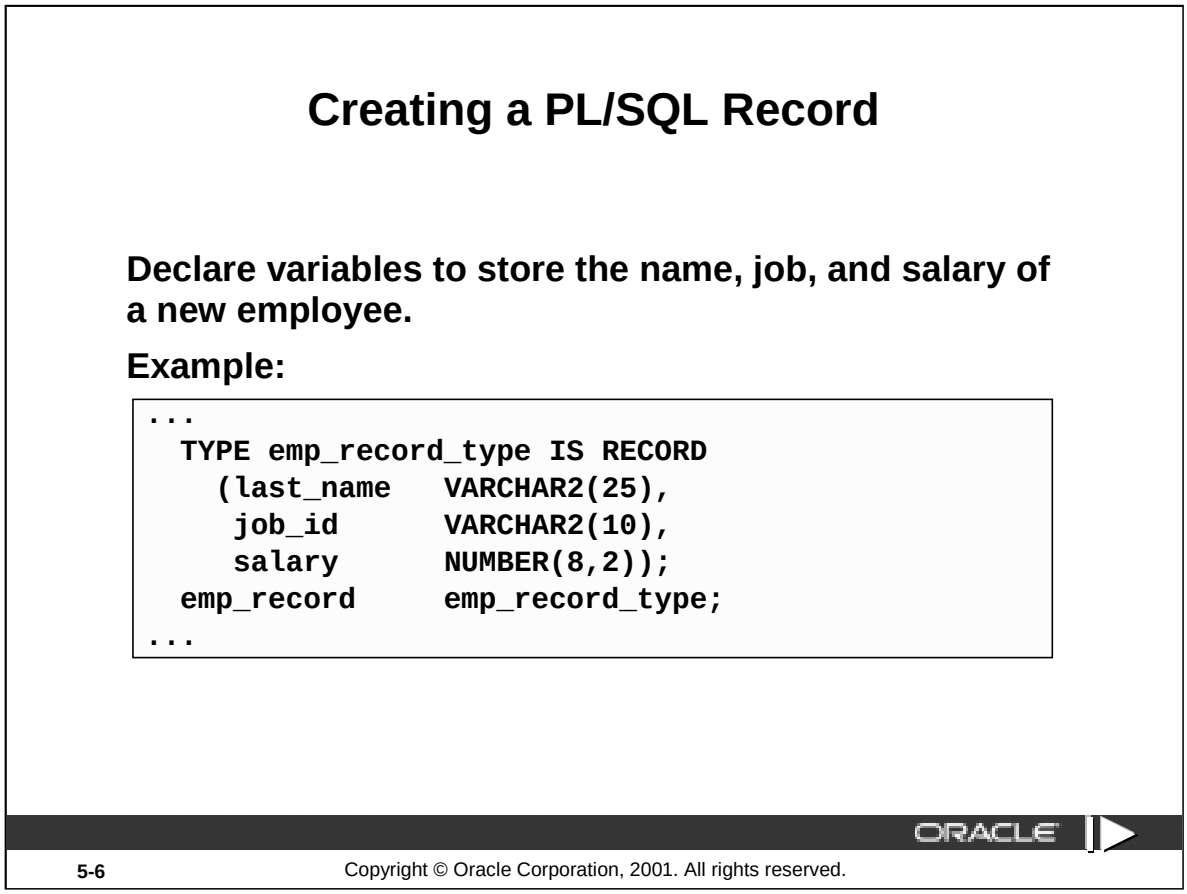
Defining and Declaring a PL/SQL Record

To create a record, you define a RECORD type and then declare records of that 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 NOT NULL fields.



Creating a PL/SQL Record

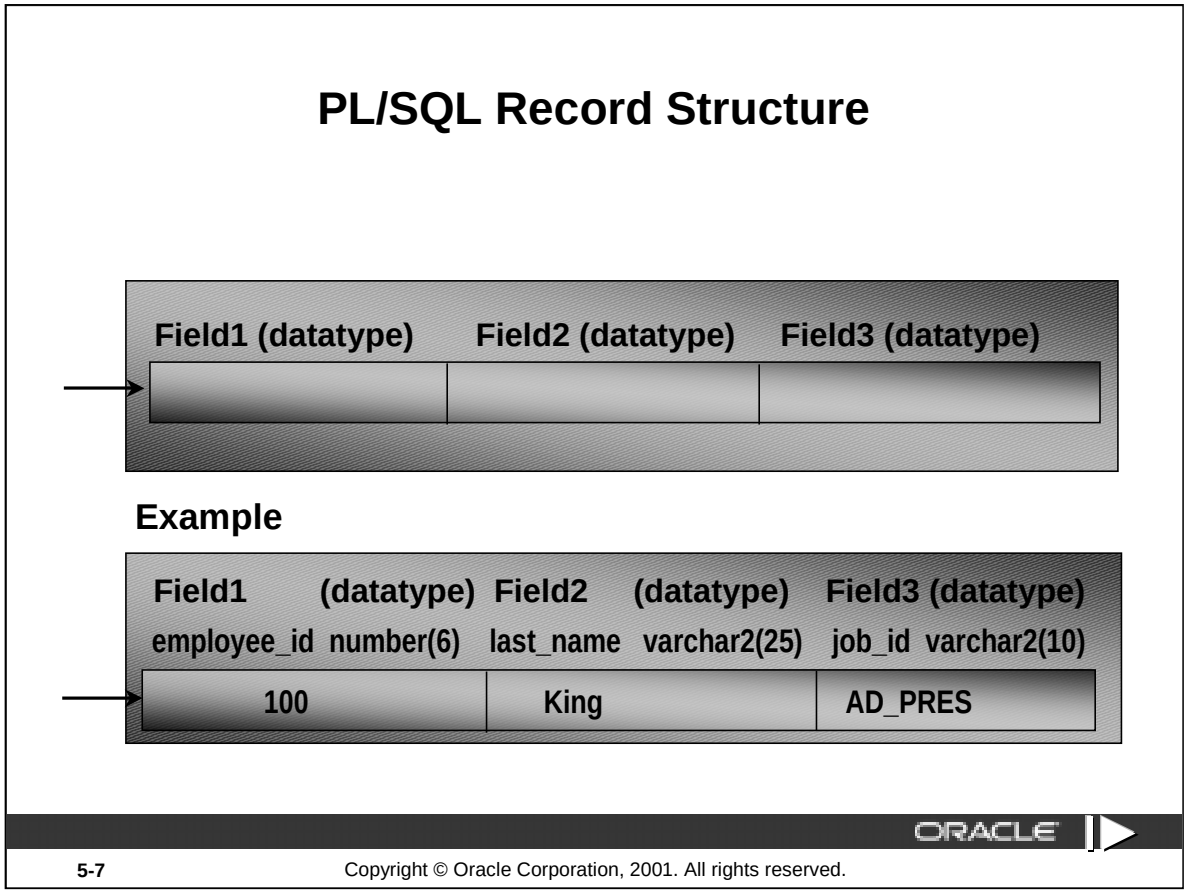
Field declarations are like variable declarations. Each field has a unique name and a specific data type. There are no predefined data types for PL/SQL records, as there are for scalar variables. Therefore, you must create the record type first and then declare an identifier using that type.

In the example on the slide, a `EMP_RECORD_TYPE` record type is defined to hold the values for the `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, fields declared as `NOT NULL` must be initialized.



PL/SQL Record Structure

Fields in a record are accessed by name. To reference or initialize an individual field, use dot notation and the following syntax:

```
record_name.field_name
```

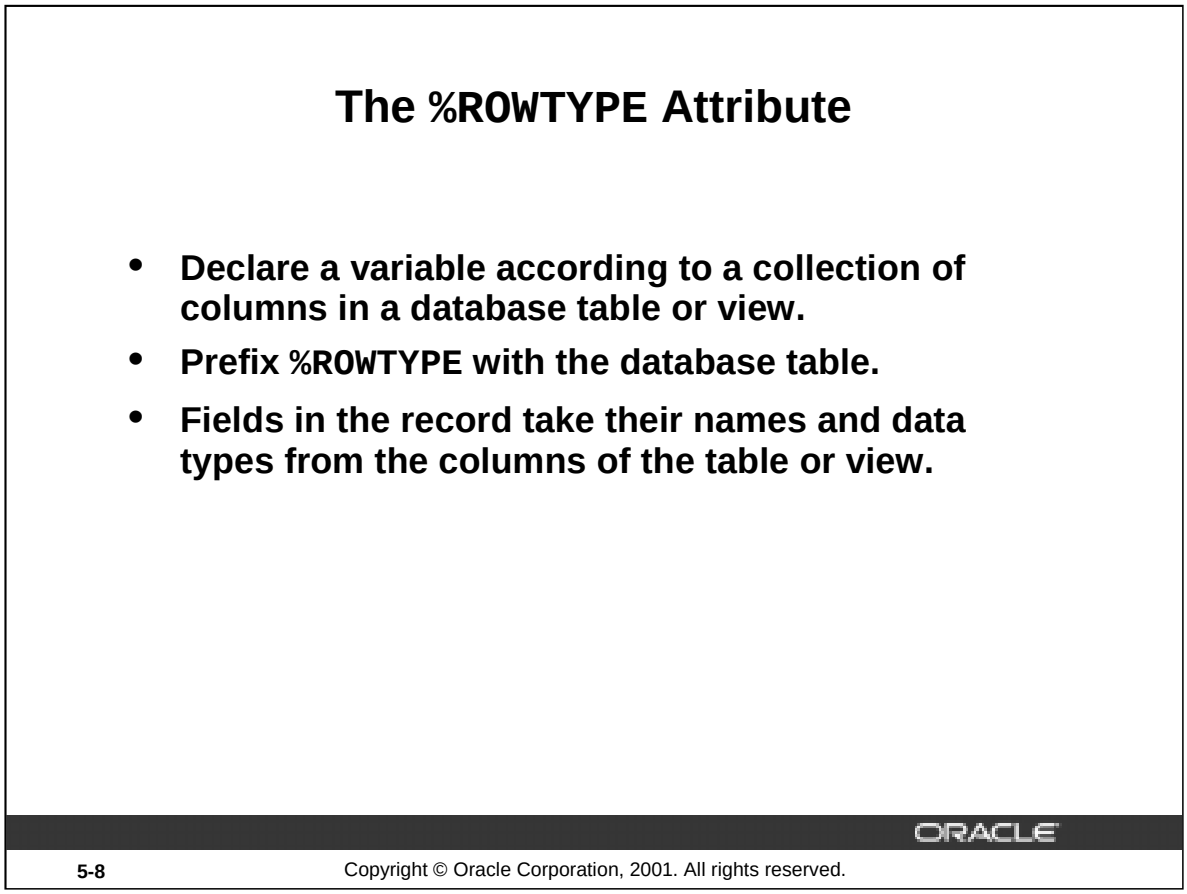
For example, you reference the job_id field in the emp_record record as follows:

```
emp_record.job_id ...
```

You can then assign a value to the record field as follows:

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

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



Declaring Records with the %ROWTYPE Attribute

To declare a record based on a collection of columns in a database table or view, you use the %ROWTYPE attribute. 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.

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

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

The record, emp_record, will have 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))
```

Declaring Records with the %ROWTYPE Attribute (continued)

Syntax

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

where: *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.

To reference an individual field, you use dot notation and the following syntax:

```
record_name.field_name
```

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

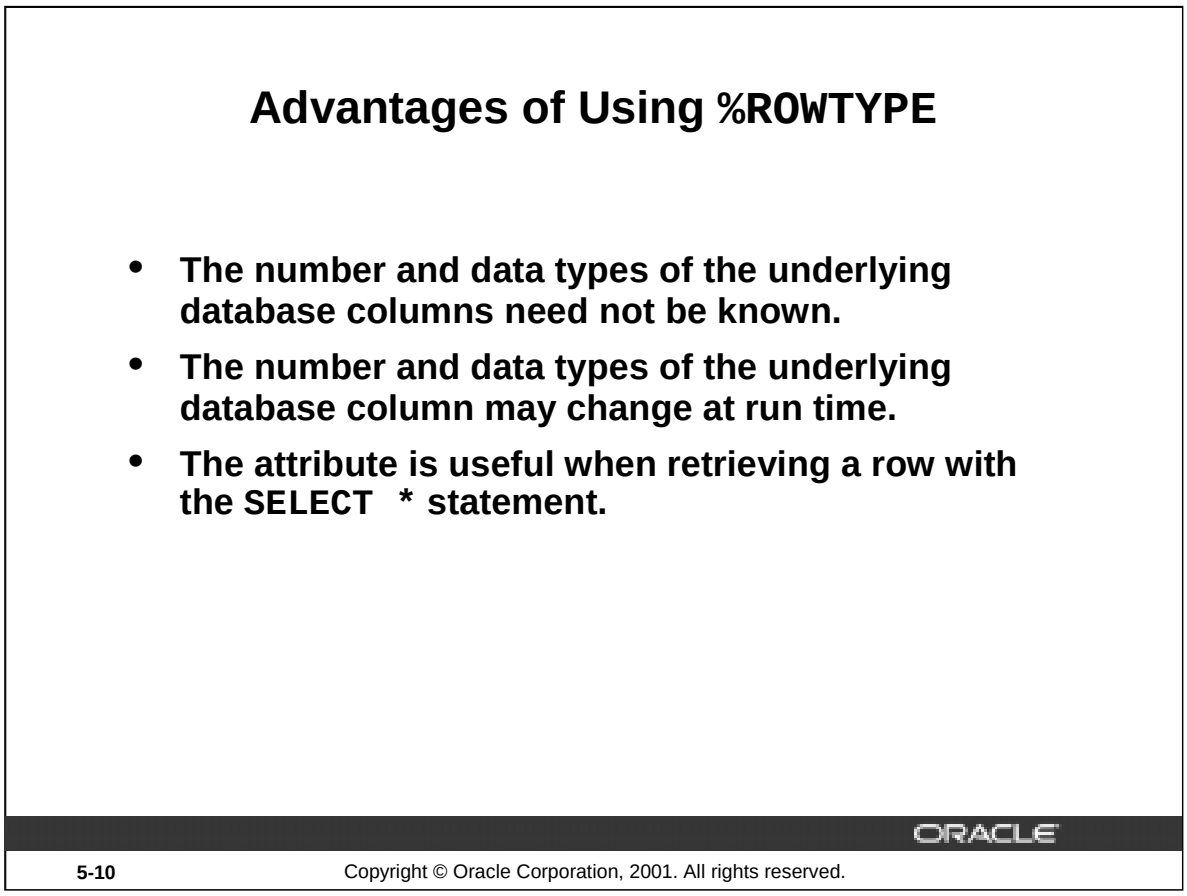
```
emp_record.commission_pct
```

You can then assign a value to the record field as follows:

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

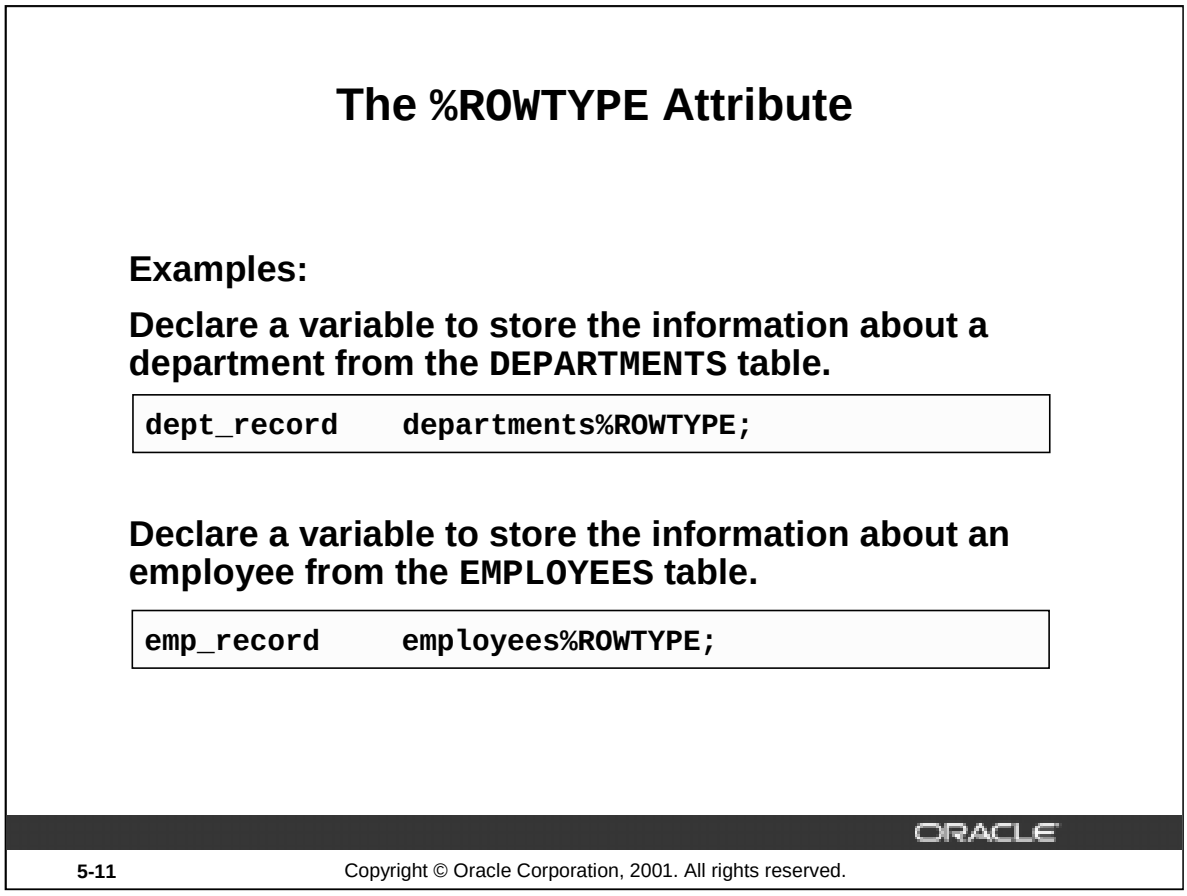
Assigning Values to Records

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 they have the same data type. A user-defined record and a `%ROWTYPE` record *never* have the same data type.



Advantages of Using %ROWTYPE

The advantages of using the %ROWTYPE attribute are listed on the slide. Use the %ROWTYPE attribute when you are not sure about the structure of the underlying database table. Using this attribute also ensures that the data types of the variables declared using this attribute change dynamically, in case the underlying table is altered. This 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.



The %ROWTYPE Attribute

The first declaration on the slide creates a record with the same field names and field data types as a row in the DEPARTMENTS table. The fields are DEPARTMENT_ID, DEPARTMENT_NAME, MANAGER_ID, and LOCATION_ID. The second declaration creates a record with the same field names, field data types, and order as a row in the EMPLOYEES table. The fields are EMPLOYEE_ID, FIRST_NAME, LAST_NAME, EMAIL, PHONE_NUMBER, HIRE_DATE, JOB_ID, SALARY, COMMISSION_PCT, MANAGER_ID, DEPARTMENT_ID.

The %ROWTYPE Attribute (continued)

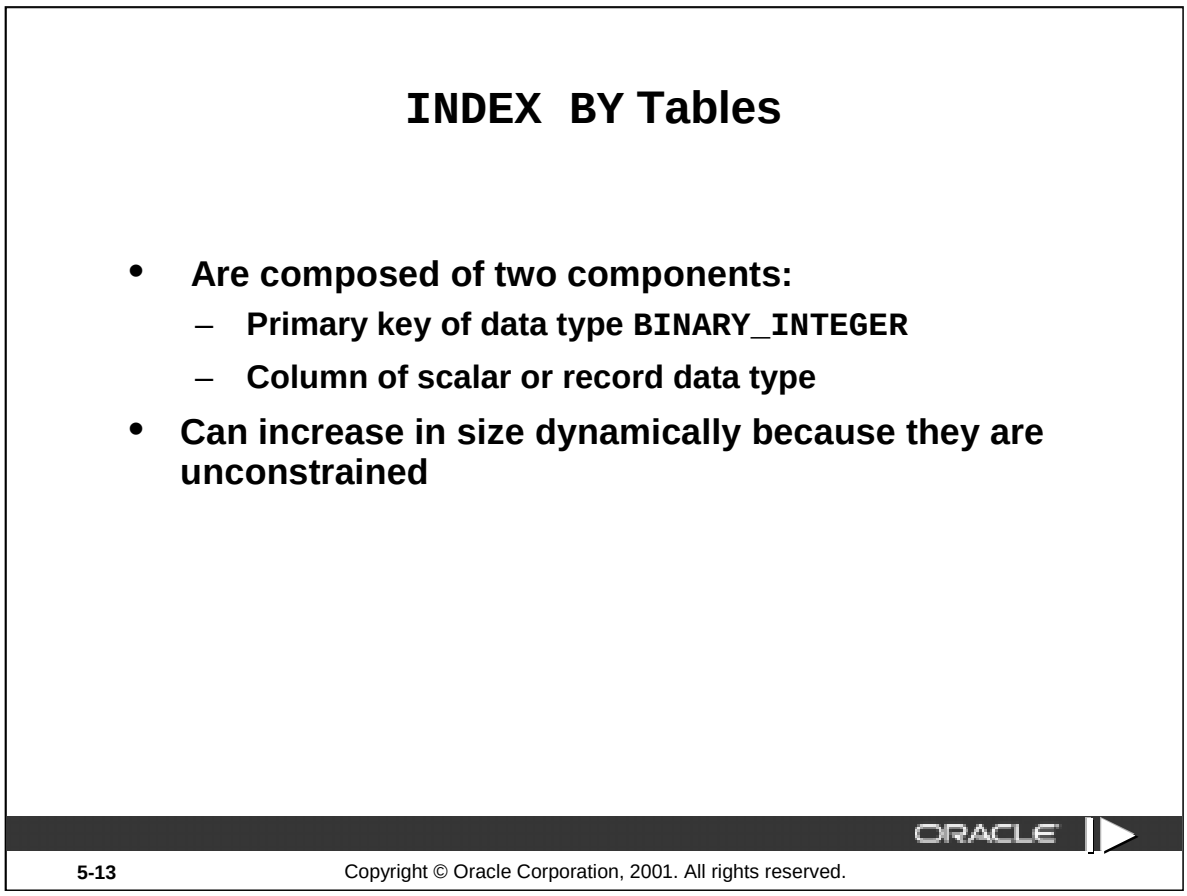
In the following example, an employee is retiring. Information about a retired employee is added to a table that holds information about retired employees. The user supplies the employee's number. The record of the employee specified by the user is retrieved from the EMPLOYEES and stored into the emp_rec variable, which is declared using the %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);
   COMMIT;
  END;
  /
```

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

```
SELECT * FROM RETIRED_EMPS;
```

EMPNO	ENAME	JOB	MGR	HIREDATE	LEAVEDATE	SAL	COMM	DEPTNO
124	Mourgos	ST_MAN	100	16-NOV-99	30-APR-01	5800		50

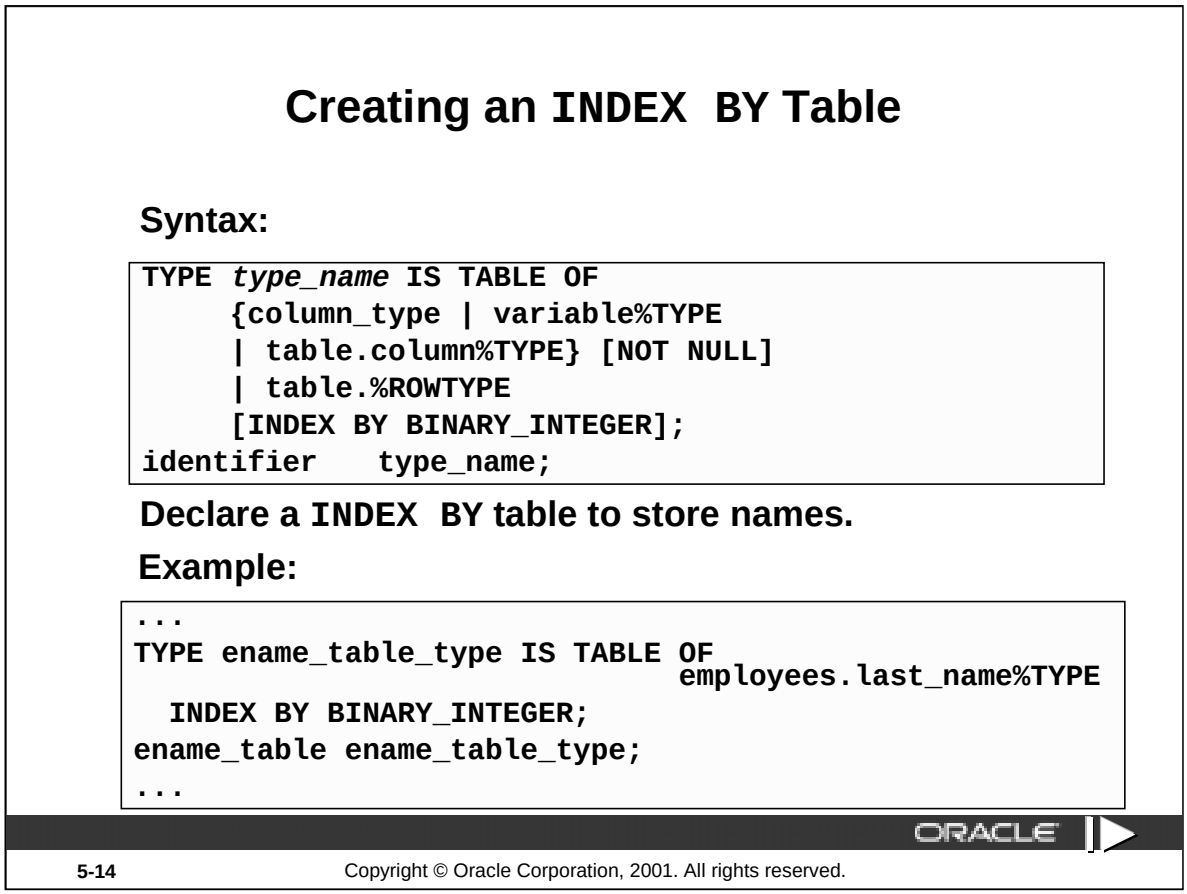


INDEX BY Tables

Objects of the TABLE type are called INDEX BY tables. They are modeled as (but not the same as) database tables. INDEX BY tables use a primary key to provide you with array-like access to rows.

A INDEX BY table:

- Is similar to an array
- Must contain two components:
  - A primary key of data type BINARY_INTEGER that indexes the INDEX BY table
  - A column of a scalar or record data type, which stores the INDEX BY table elements
- Can increase dynamically because it is unconstrained



Creating a INDEX BY Table

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

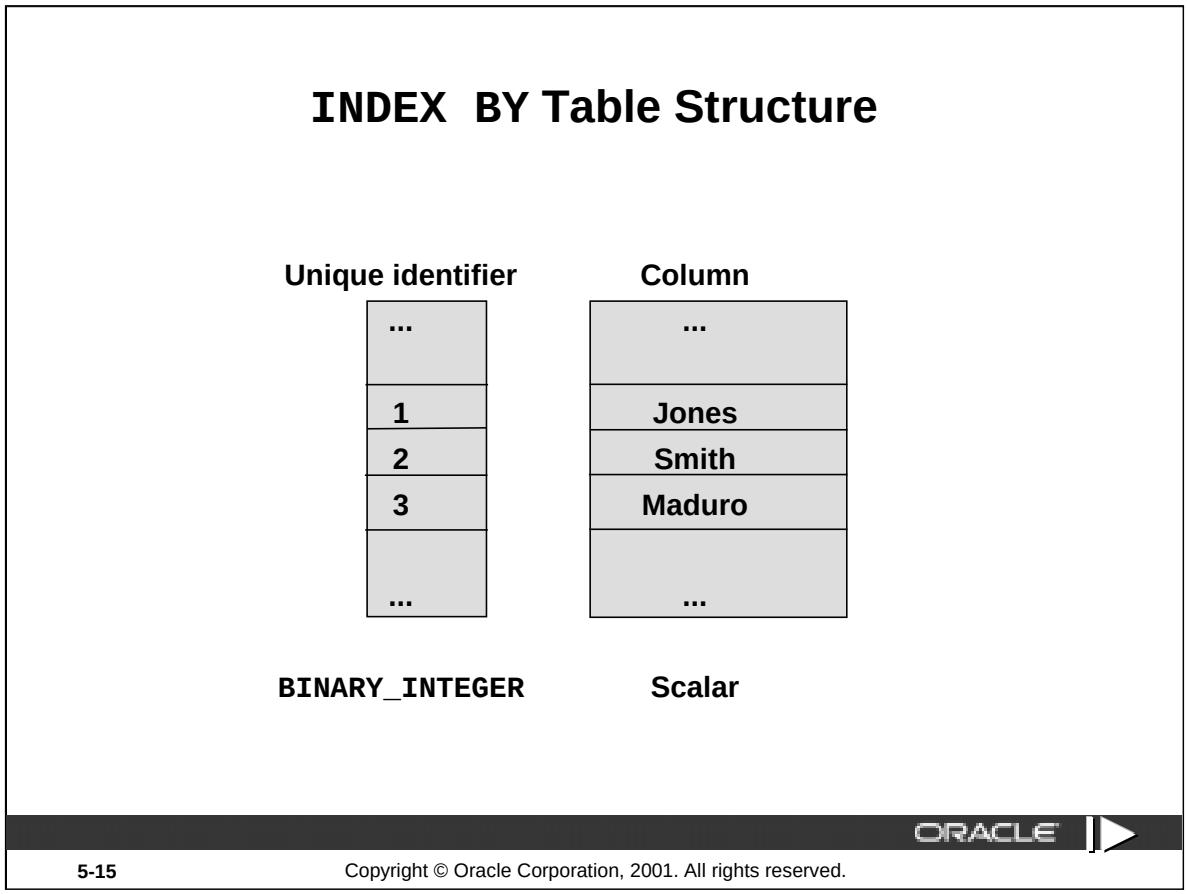
1. Declare a TABLE data type.
2. Declare a variable of that data type.

In the syntax:

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

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 initially sparse. That enables you, for example, to store reference data in an INDEX-BY table using a numeric primary key as the index.



INDEX BY Table Structure

Like the size of a database table, the size of a INDEX BY table is unconstrained. That is, the number of rows in a 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 one 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 type 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 no values. An explicit executable statement is required to initialize (populate) the INDEX BY table.

Creating an INDEX BY Table

```
DECLARE
  TYPE ename_table_type IS TABLE OF
       employees.last_name%TYPE
       INDEX BY BINARY_INTEGER;
  TYPE hiredate_table_type IS TABLE OF DATE
       INDEX BY BINARY_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;
/
```

Referencing an INDEX BY Table

Syntax:

INDEX_BY_table_name(primary_key_value)

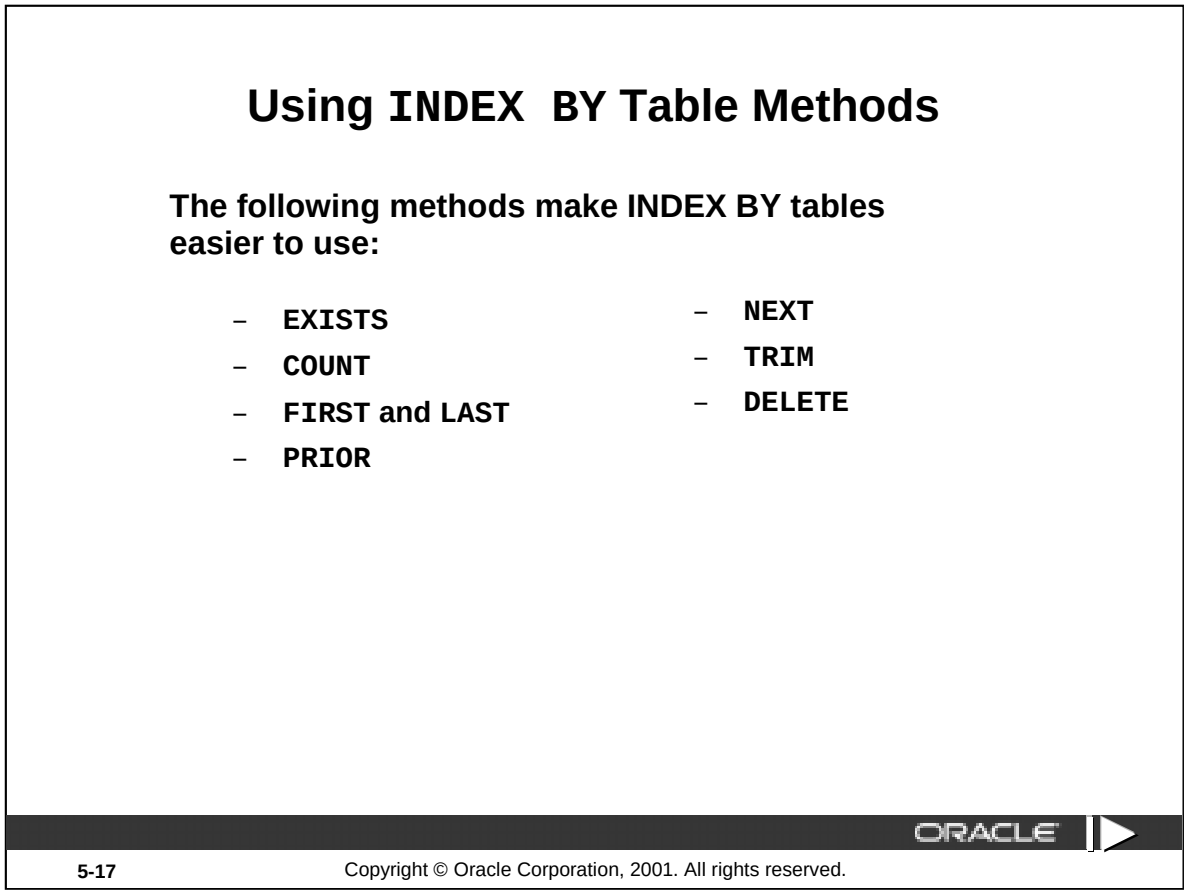
where: *primary_key_value* belongs to type BINARY_INTEGER.

Reference the third row in an INDEX BY table ENAME_TABLE:

```
ename_table(3) ...
```

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

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

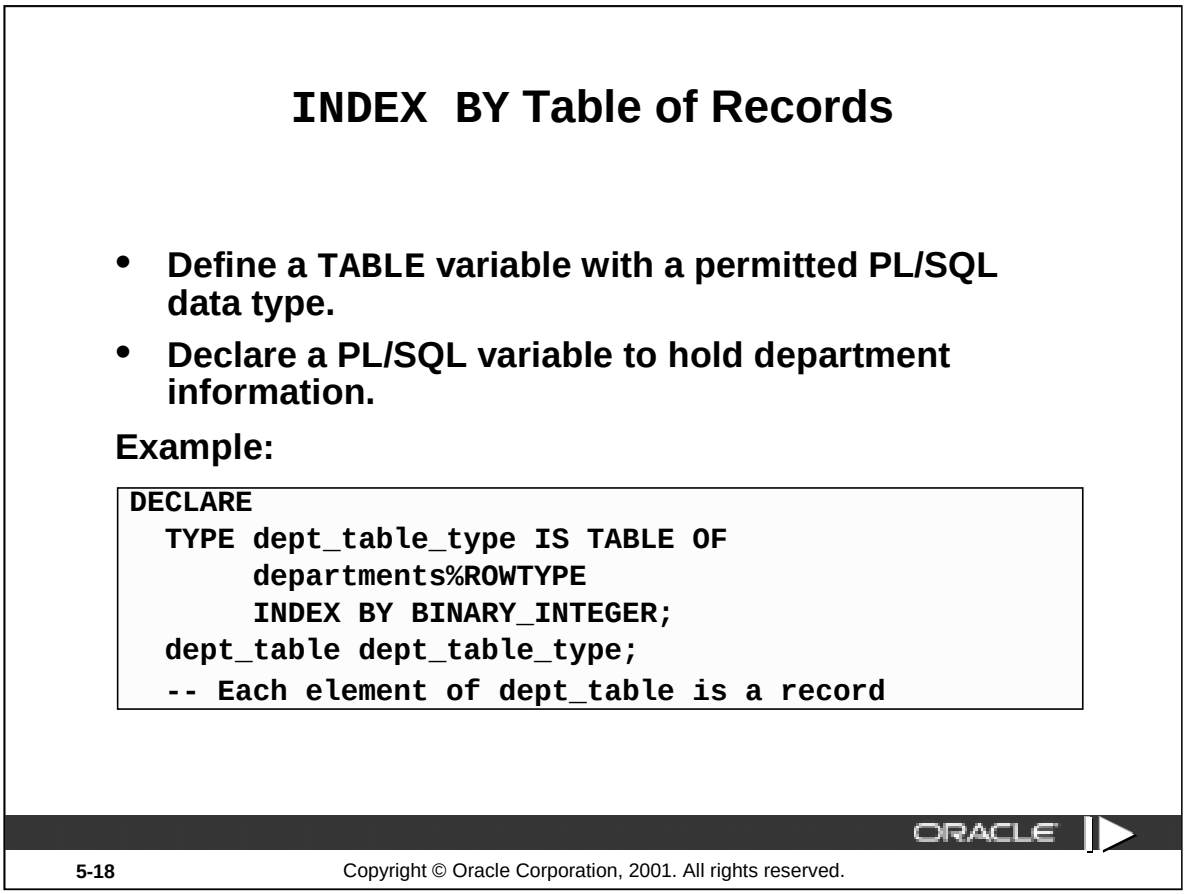


Using `INDEX BY` Table Methods

A `INDEX BY` table method is a built-in procedure or function that operates on tables and is called 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
TRIM	TRIM removes one element from the end of a PL/SQL table. TRIM(n) removes *n* elements from the end of 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

At a given point of time, a INDEX BY table can store only the details of any one of the columns of a database table. There is always a necessity to store all the columns retrieved by a query. The INDEX BY table of records offer a solution to this. Because only one table definition is needed to hold information about all of 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 given on the slide, you can refer to fields in the DEPT_TABLE record because each element of this table is a record.

Syntax:

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

Example:

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

LOCATION_ID represents a field in DEPT_TABLE.

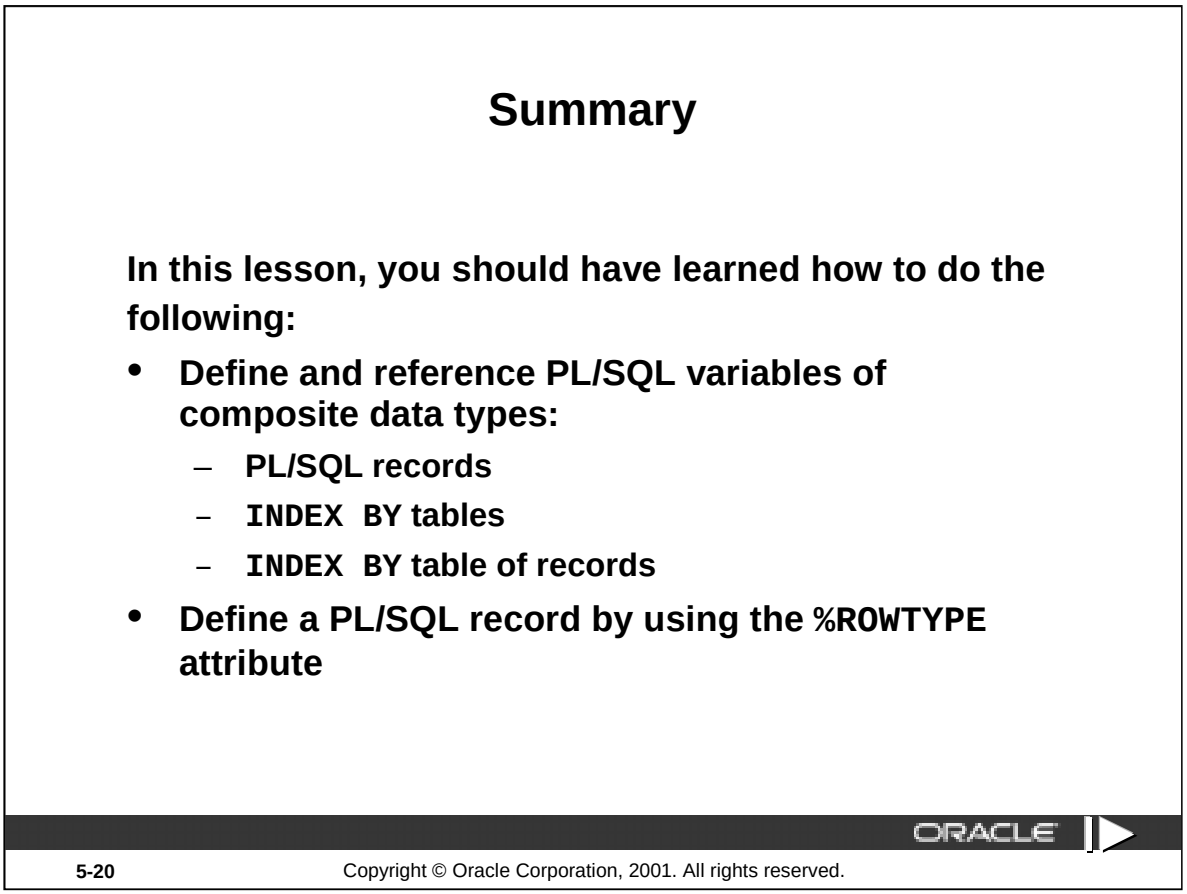
Note: You can use the %ROWTYPE attribute to declare a record that represents a row in a database table. The difference between the %ROWTYPE attribute and the composite data type RECORD is that RECORD allows you to specify the data types of fields in the record or to declare fields of your own.

Example of INDEX BY Table of Records

```
SET SERVEROUTPUT ON
DECLARE
   TYPE emp_table_type is table of
   employees%ROWTYPE INDEX BY BINARY_INTEGER;
   my_emp_table     emp_table_type;
   v_count   NUMBER(3):= 104;
BEGIN
  FOR i IN 100..v_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;
```

Example INDEX BY Table of Records

The example on the slide declares a INDEX BY table of records emp_table_type to temporarily store the details of the employees whose EMPLOYEE_ID lies 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 information regarding the last names from the INDEX BY table. Observe the use of the FIRST and LAST methods in the example.



Summary

A PL/SQL record is a collection of individual fields that represent a row in a 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.

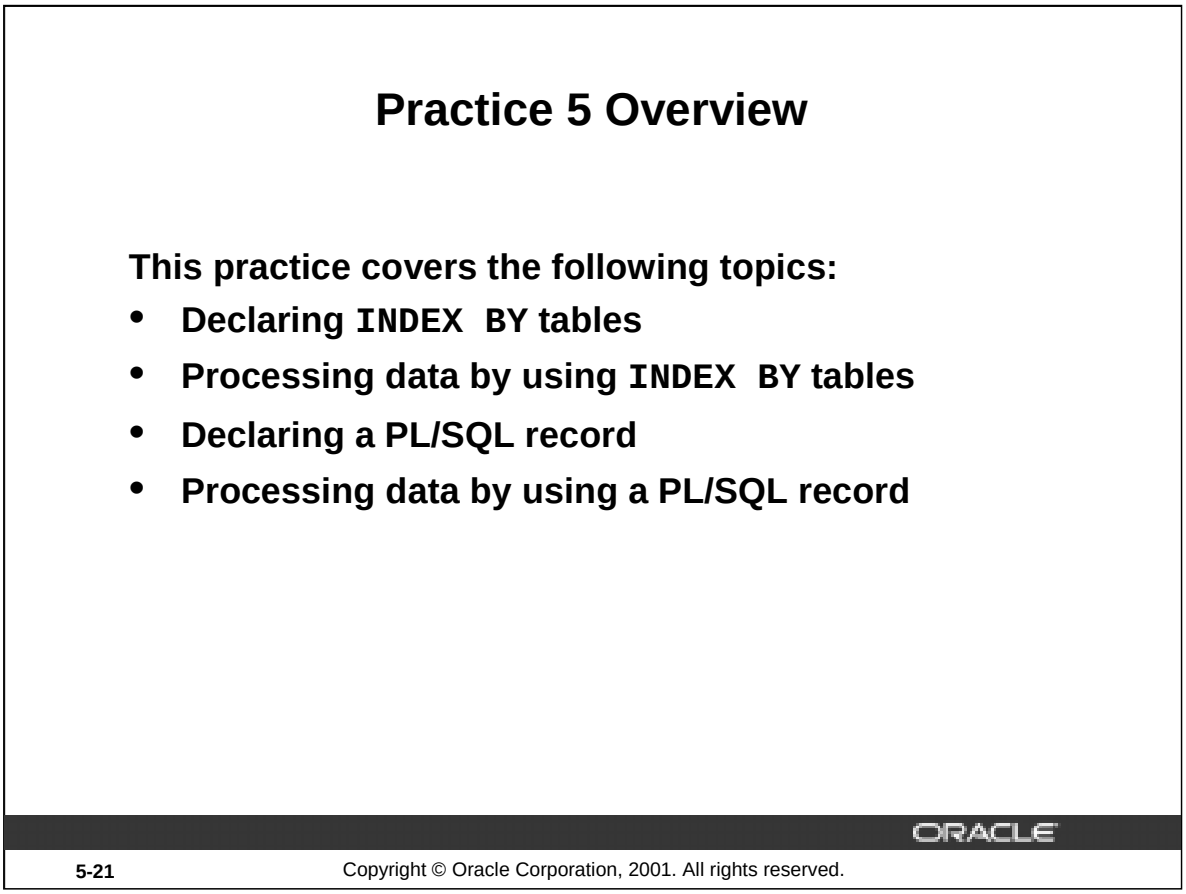
Like PL/SQL records, the table is another composite data type. `INDEX BY` tables are objects of a `TABLE` type and look similar to database tables but with a slight difference. `INDEX BY` tables use a primary key to give you array-like access to rows. The size of a `INDEX BY` table is unconstrained. `INDEX BY` tables can have one column and a primary key, neither of which can be named. The column can have any data type, but the primary key must be of the `BINARY_INTEGER` type.

A `INDEX BY` table of records enhances the functionality of `INDEX BY` tables, because only one table definition is required to hold information about all the fields.

The following collection methods help generalize code, make collections easier to use, and make your applications easier to maintain:

`EXISTS`, `COUNT`, `LIMIT`, `FIRST` and `LAST`, `PRIOR` and `NEXT`, `TRIM` , and `DELETE`

The `%ROWTYPE` is used to declare a compound variable whose type is the same as that of a row of a database table.



Practice 5 Overview

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

Practice 5

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 provide the country ID. Pass the value to the PL/SQL block through a *i*SQL*Plus substitution variable.
   c. Use `DBMS_OUTPUT.PUT_LINE` to print selected information about the country. A sample output is shown below.

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

   d. Execute and test the PL/SQL block for the countries with the IDs CA, DE, UK, US.

2. Create a PL/SQL block to retrieve the name of each department from the `DEPARTMENTS` table and print each department name on the screen, incorporating an `INDEX BY` table. Save the code in a file called `p5_q2.sql` by clicking the `Save Script` button. Save the script with a `.sql` extension.
   a. Declare an `INDEX BY` table, `MY_DEPT_TABLE`, to temporarily store the name of the departments.
   b. Using a loop, retrieve the name of all departments currently in the `DEPARTMENTS` table and store them in the `INDEX BY` table. Use the following table to assign the value for `DEPARTMENT_ID` based on the value of the counter used in the loop.

COUNTER	DEPARTMENT_ID
1	10
2	20
3	50
4	60
5	80
6	90
7	110

   c. Using another loop, retrieve the department names from the `INDEX BY` table and print them to the screen, using `DBMS_OUTPUT.PUT_LINE`. The output from the program is shown on the next page.

Practice 5 (continued)

```
Administration
Marketing
Shipping
IT
Sales
Executive
Accounting
Accounting
Accounting
Accounting
Accounting
Accounting
Accounting
Accounting
Accounting
Accounting
Accounting
Accounting
Accounting
Accounting
Accounting
Accounting
Accounting
Accounting
Accounting
Accounting
Accounting
Accounting
PL/SQL procedure successfully completed.
```

Practice 5 (continued)

If you have time, complete the following exercise.

3. Modify the block you created in practice 2 to retrieve all information about each department from the DEPARTMENTS table and print the information to the screen, incorporating an INDEX BY table of records.

   a. Declare an INDEX BY table, MY_DEPT_TABLE, to temporarily store the number, name, and location of all the departments.

   b. Using a loop, retrieve all department information currently in the DEPARTMENTS table and store it in the INDEX BY table. Use the following table to assign the value for DEPARTMENT_ID based on the value of the counter used in the loop. Exit the loop when the counter reaches the value 7.

COUNTER	DEPARTMENT_ID
1	10
2	20
3	50
4	60
5	80
6	90
7	110

   c. Using another loop, retrieve the department information from the INDEX BY table and print it to the screen, using DBMS_OUTPUT.PUT_LINE. A sample output is shown.

```
Department Number: 10 Department Name: Administration Manager Id: 200 Location Id: 1400
Department Number: 20 Department Name: Marketing Manager Id: 201 Location Id: 1800
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: 80 Department Name: Sales Manager Id: 145 Location Id: 2500
Department Number: 90 Department Name: Executive Manager Id: 100 Location Id: 1700
Department Number: 110 Department Name: Accounting Manager Id: 205 Location Id: 1700
PL/SQL procedure successfully completed.
```

6

Writing Explicit Cursors

ORACLE

Objectives

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

- **Distinguish between an implicit and an explicit cursor**
- **Discuss when and why to use an explicit cursor**
- **Use a PL/SQL record variable**
- **Write a cursor FOR loop**

ORACLE

Lesson Aim

In this lesson, you learn the difference between implicit and explicit cursors. You also learn when and why to use an explicit cursor. You may need to use a multiple-row SELECT statement in PL/SQL to process many rows. To accomplish this, you declare and control explicit cursors.

About Cursors

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

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

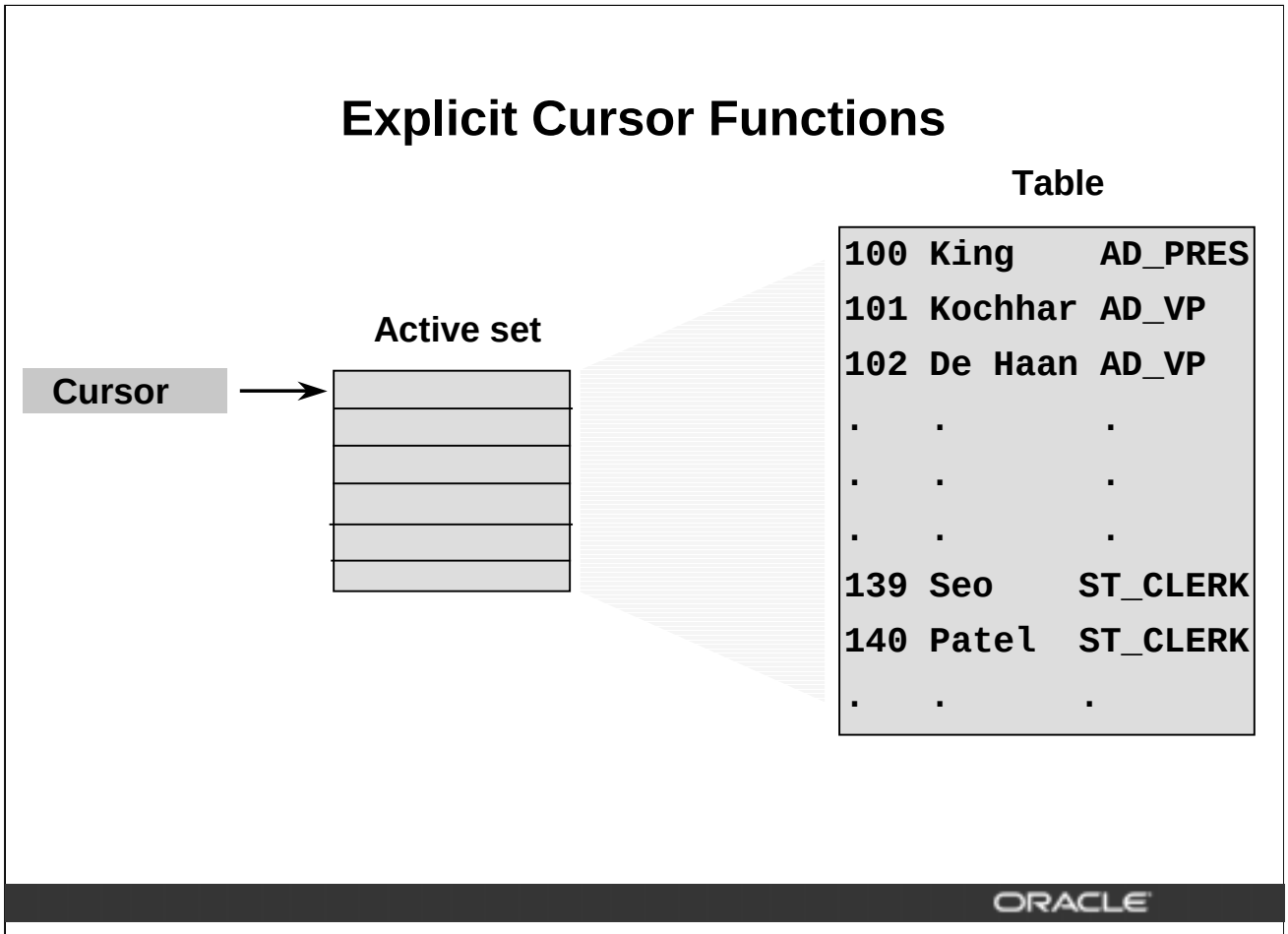
ORACLE

Implicit and Explicit Cursors

The Oracle server uses work areas, called private SQL areas, to execute SQL statements and to store processing information. You can use PL/SQL cursors to name a private SQL area and access its stored information.

Cursor Type	Description
Implicit	Implicit cursors are declared by PL/SQL implicitly for all DML and PL/SQL SELECT statements, including queries that return only one row.
Explicit	For queries that return more than one row, explicit cursors are declared and named 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 not associated with an explicitly declared cursor. PL/SQL allows you to refer to the most recent implicit cursor as the *SQL* cursor.



Explicit Cursors

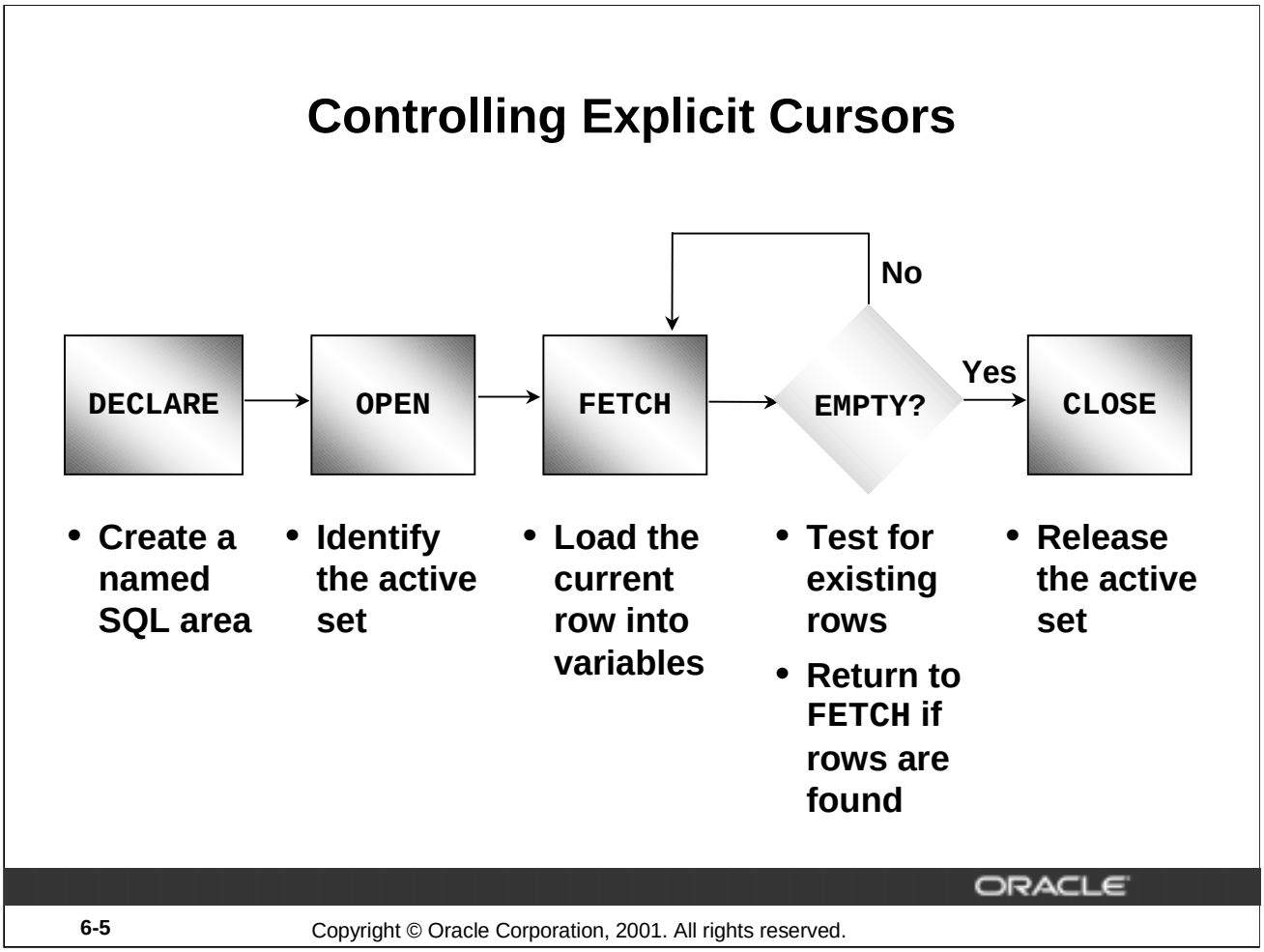
Use explicit cursors to individually process each row returned by a multiple-row SELECT statement.

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 on the slide shows how an explicit cursor "points" to the *current row* in the active set. This allows your program to process the rows one at a time.

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.

Explicit cursor functions:

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



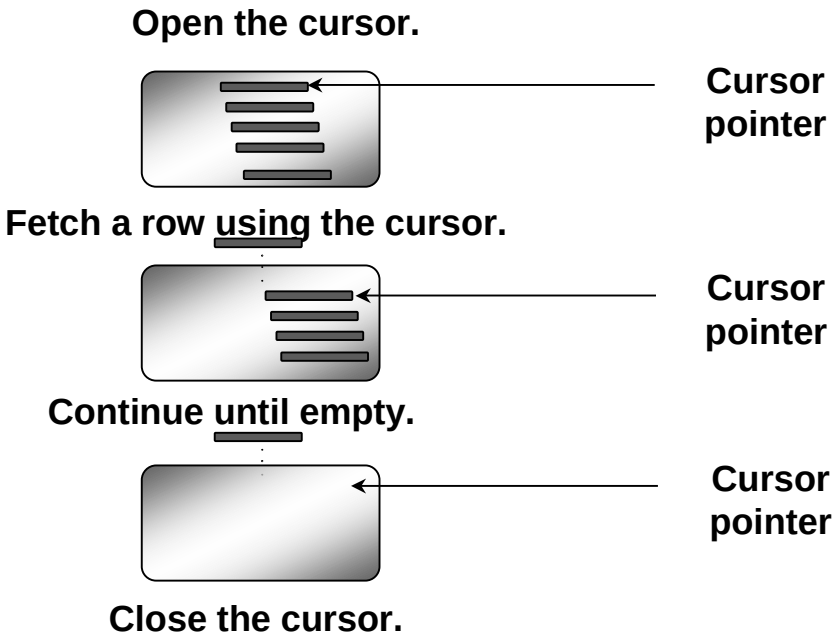
Explicit Cursors (continued)

Now that you have a conceptual understanding of cursors, review the steps to use them. The syntax for each step can be found on the following pages.

Controlling Explicit Cursors

1. Declare the cursor by naming it and defining the structure of the query to be performed within 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 on the slide, after each fetch you test the cursor for any existing row. If there are no more rows to process, then 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



ORACLE

Explicit Cursors (continued)

You use the OPEN, FETCH, and CLOSE statements to control a cursor. The OPEN statement executes the query associated with the cursor, identifies the result set, and positions the cursor before the first row.

The FETCH statement retrieves the current row and advances the cursor to the next row. When the last row has been processed, the CLOSE statement disables the cursor.

Declaring the Cursor

Syntax:

```
CURSOR cursor_name IS
     select_statement;
```

- **Do not include the `INTO` clause in the cursor declaration.**
- **If processing rows in a specific sequence is required, use the `ORDER BY` clause in the query.**

ORACLE

Declaring the Cursor

Use the `CURSOR` statement to declare an explicit cursor. You can reference variables within the query, but you must declare them before the `CURSOR` statement.

In the syntax:

cursor_name	is a PL/SQL identifier.
select_statement	is a `SELECT` statement without an `INTO` clause.

Note:

- Do not include the `INTO` clause in the cursor declaration because it appears later in the `FETCH` statement.
- The cursor can be any valid `ANSI SELECT` statement, to include joins, and so on.

Declaring the Cursor

Example:

```
DECLARE
  CURSOR emp_cursor IS
    SELECT employee_id, last_name
    FROM   employees;

  CURSOR dept_cursor IS
    SELECT *
    FROM   departments
    WHERE  location_id = 170;
BEGIN
  ...
```

ORACLE

Declaring the Cursor (continued)

In the example on the slide, the cursor `emp_cursor` is declared to retrieve the `EMPLOYEE_ID` and `LAST_NAME` columns from the `EMPLOYEES` table. Similarly, the cursor `DEPT_CURSOR` is declared to retrieve all the details for the department with the `LOCATION_ID 170`.

```
DECLARE
  v_empno         employees.employee_id%TYPE;
  v_ename         employees.last_name%TYPE;
  CURSOR emp_cursor IS
    SELECT employee_id, last_name
    FROM   employees;
BEGIN
. . .
```

Fetching the values retrieved by the cursor into the variables declared in the `DECLARE` section is covered later in this lesson.

Opening the Cursor

Syntax:

```
OPEN  cursor_name;
```

- **Open the cursor to execute the query and identify the active set.**
- **If the query returns no rows, no exception is raised.**
- **Use cursor attributes to test the outcome after a fetch.**

ORACLE

OPEN Statement

The OPEN statement executes the query associated with the cursor, identifies the result set, and positions the cursor before the first row.

In the syntax:

cursor_name is the name of the previously declared cursor.

OPEN is an executable statement that performs the following operations:

1. Dynamically allocates memory for a context area that eventually contains crucial processing information.
2. Parses the SELECT statement.
3. Binds the input variables—sets the value 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.
5. Positions the pointer just before the first row in the active set.

For cursors declared using the FOR UPDATE clause, the OPEN statement also locks those rows. The FOR UPDATE clause is discussed in a later lesson.

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 cursor after a fetch using the SQL%ROWCOUNT cursor attribute.

Fetching Data from the Cursor

Syntax:

```
FETCH cursor_name INTO [variable1, variable2, ...]
                       | record_name];
```

- **Retrieve the current row values into variables.**
- **Include the same number of variables.**
- **Match each variable to correspond to the columns positionally.**
- **Test to see whether the cursor contains rows.**

ORACLE

FETCH Statement

The FETCH statement retrieves the rows in the active set one at a time. After each fetch, the cursor advances to the next row in the active set.

In the syntax:

cursor_name	is the name of the previously declared cursor.
variable	is an output variable to store the results.
record_name	is the name of the record in which the retrieved data is stored. (The record variable can be declared using the %ROWTYPE attribute.)

Guidelines:

- Include the same number of variables in the INTO clause of the FETCH statement as 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.

Note: 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 identified set.

Fetching Data from the Cursor

Example:

```
LOOP
  FETCH emp_cursor INTO v_empno,v_ename;
  EXIT WHEN ...;
  ...
    -- Process the retrieved data
  …
END LOOP;
```

ORACLE

FETCH Statement (continued)

You use the FETCH statement to retrieve the current row values into output variables. After the fetch, you can manipulate the data in the variables. For each column value returned by the query associated with the cursor, there must be a corresponding variable in the INTO list. Also, their data types must be compatible.

Retrieve the first 10 employees one by one.

```
SET SERVEROUTPUT ON
    DECLARE
     v_empno  employees.employee_id%TYPE;
     v_ename  employees.last_name%TYPE;
     CURSOR   emp_cursor IS
       SELECT employee_id, last_name
       FROM   employees;
   BEGIN
     OPEN emp_cursor;
     FOR i IN 1..10 LOOP
       FETCH emp_cursor INTO v_empno, v_ename;
       DBMS_OUTPUT.PUT_LINE (TO_CHAR(v_empno)
       ||'     '|| v_ename);
     END LOOP;
   END ;
```

Closing the Cursor

Syntax:

```
CLOSE     cursor_name;
```

- **Close the cursor after completing the processing of the rows.**
- **Reopen the cursor, if required.**
- **Do not attempt to fetch data from a cursor after it has been closed.**

ORACLE

CLOSE Statement

The CLOSE statement disables the cursor, and the active set becomes undefined. Close the cursor after completing the processing of the SELECT statement. This step allows the cursor to be reopened, if required. Therefore, you can establish an active set several times.

In the syntax:

cursor_name is the name of the previously declared cursor.

Do not attempt to fetch data from a cursor after it has been closed, or the INVALID_CURSOR exception will be raised.

Note: The CLOSE statement releases the context area.

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 to the number of open cursors per user, which is determined by the OPEN_CURSORS parameter in the database parameter file. OPEN_CURSORS = 50 by default.

```
    OPEN emp_cursor
    FOR i IN 1..10 LOOP
      FETCH emp_cursor INTO v_empno, v_ename;
      ...
    END LOOP;
    CLOSE emp_cursor;
  END;
```

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 data manipulation statement.

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

The %ISOPEN Attribute

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

Example:

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

ORACLE

The %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 for the following:
  - To retrieve an exact number of rows
  - Fetch the rows in a numeric FOR loop
  - Fetch the rows in a simple loop and determine when to exit the loop.

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

Controlling Multiple Fetches

- **Process several rows from an explicit cursor using a loop.**
- **Fetch a row with each iteration.**
- **Use explicit cursor attributes to test the success of each fetch.**

ORACLE

Controlling Multiple Fetches from Explicit Cursors

To process several rows from an explicit cursor, you typically define a loop to perform a fetch on each iteration. Eventually all rows in the active set are processed, and an unsuccessful fetch sets the %NOTFOUND attribute to TRUE. Use the explicit cursor attributes to test the success of each fetch before any further references are made to the cursor. If you omit an exit criterion, an infinite loop results.

For more information, see *PL/SQL User's Guide and Reference*, "Interaction With Oracle."

The %NOTFOUND and %ROWCOUNT Attributes

- Use the %ROWCOUNT cursor attribute to retrieve an exact number of rows.
- Use the %NOTFOUND cursor attribute to determine when to exit the loop.

ORACLE

The %NOTFOUND and %ROWCOUNT Attributes

%NOTFOUND

%NOTFOUND is the logical opposite of %FOUND. %NOTFOUND yields FALSE if the last fetch returned a row, or TRUE if the last fetch failed to return a row. In the following example, you use %NOTFOUND to exit a loop when FETCH fails to return a row:

```
LOOP
   FETCH c1 INTO my_ename, my_sal, my_hiredate;
   EXIT WHEN c1%NOTFOUND;
   ...
END LOOP;
```

Before the first fetch, %NOTFOUND evaluates to NULL. So, if FETCH never executes successfully, the loop is never exited. That is because the EXIT WHEN statement executes only if its WHEN condition is true. To be safe, use the following EXIT statement instead:

```
EXIT WHEN c1%NOTFOUND OR c1%NOTFOUND IS NULL;
```

If a cursor is not open, referencing it with %NOTFOUND raises INVALID_CURSOR.

The %NOTFOUND and %ROWCOUNT Attributes (continued)

%ROWCOUNT

When its cursor or cursor variable is opened, %ROWCOUNT is zeroed. Before the first fetch, %ROWCOUNT yields 0. Thereafter, it yields the number of rows fetched so far. The number is incremented if the last fetch returned a row. In the next example, you use %ROWCOUNT to take action if more than ten rows have been fetched:

```
LOOP
   FETCH c1 INTO my_ename, my_deptno;
   IF c1%ROWCOUNT > 10 THEN
      ...
   END IF;
   ...
END LOOP;
```

If a cursor is not open, referencing it with %ROWCOUNT raises INVALID_CURSOR.

Example

```
DECLARE
      v_empno  employees.employee_id%TYPE;
      v_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 v_empno, v_ename;
        EXIT WHEN emp_cursor%ROWCOUNT > 10 OR
                          emp_cursor%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE (TO_CHAR(v_empno)
                              ||'  '|| v_ename);
      END LOOP;
      CLOSE emp_cursor;
END ;
```

ORACLE

Example

The example on the slide retrieves the first ten employees one by one.

Note: Before the first fetch, `%NOTFOUND` evaluates to `NULL`. So if `FETCH` never executes successfully, the loop is never exited. That is because the `EXIT WHEN` statement executes only if its `WHEN` condition is true. To be safe, use the following `EXIT` statement:

```
EXIT WHEN emp_cursor%NOTFOUND OR emp_cursor%NOTFOUND IS NULL;
```

If using `%ROWCOUNT`, add a test for no rows in the cursor by using the `%NOTFOUND` attribute, because the row count is not incremented if the fetch does not retrieve any rows.

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;
  emp_record   emp_cursor%ROWTYPE;
BEGIN
  OPEN emp_cursor;
  LOOP
    FETCH emp_cursor INTO emp_record;
  ...
```

```
emp_record
employee_id          last_name

100                  King
```

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.

Example

Use a cursor to retrieve employee numbers and names and populate a database table, `TEMP_LIST`, with this information.

```
DECLARE
  CURSOR emp_cursor IS
    SELECT employee_id, last_name
    FROM   employees;
  emp_record emp_cursor%ROWTYPE;
BEGIN
  OPEN emp_cursor;
  LOOP
    FETCH emp_cursor INTO emp_record;
    EXIT WHEN emp_cursor%NOTFOUND;
    INSERT INTO temp_list (empid, empname)
    VALUES (emp_record.employee_id, emp_record.last_name);
  END LOOP;
  COMMIT;
  CLOSE emp_cursor;
END;
/
```

Cursor FOR Loops

Syntax:

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

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

ORACLE

Cursor FOR Loops

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

In the syntax:

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

Guidelines

- Do not declare the record that controls the loop because it is declared implicitly.
- Test the cursor attributes during the loop, if required.
- Supply the parameters for a cursor, if required, in parentheses following the cursor name in the FOR statement. More information on cursor parameters is covered in a subsequent lesson.
- Do not use a cursor FOR loop when the cursor operations must be handled explicitly.

Note: You can define a query at the start of the loop itself. The query expression is called a SELECT substatement, and the cursor is internal to the FOR loop. Because the cursor is not declared with a name, you cannot test its attributes.

Cursor FOR Loops

Print a list of the employees who work for the sales department.

```
DECLARE
  CURSOR emp_cursor IS
    SELECT last_name, department_id
    FROM   employees;
BEGIN
  FOR emp_record IN emp_cursor LOOP
         -- implicit open and implicit fetch occur
    IF emp_record.department_id = 80 THEN
      ...
  END LOOP; -- implicit close occurs
END;
/
```

ORACLE

Example

Retrieve employees one by one and print out a list of those employees currently working in the sales department (`DEPARTMENT_ID = 80`). The example from the slide is completed below.

```
SET SERVEROUTPUT ON
DECLARE
  CURSOR emp_cursor IS
    SELECT last_name, department_id
    FROM   employees;
BEGIN
  FOR emp_record IN emp_cursor LOOP
        --implicit open and implicit fetch occur
    IF emp_record.department_id = 80 THEN
       DBMS_OUTPUT.PUT_LINE ('Employee ' || emp_record.last_name
                             || ' works in the Sales Dept. ');
    END IF;
  END LOOP;   --implicit close and implicit loop exit
END ;
/
```

Cursor FOR Loops Using Subqueries

No need to declare the cursor.

Example:

```
BEGIN
  FOR emp_record IN (SELECT last_name, department_id
                     FROM   employees) LOOP
         -- implicit open and implicit fetch occur
    IF emp_record.department_id = 80 THEN
      ...
  END LOOP; -- implicit close occurs
END;
```

ORACLE

Cursor FOR Loops Using Subqueries

When you use a subquery in a FOR loop, you do not need to declare a cursor. This example does the same thing as the one on the previous page. The complete code is given below:

```
SET SERVEROUTPUT ON
BEGIN
  FOR emp_record IN (SELECT last_name, department_id
                     FROM   employees) LOOP
        --implicit open and implicit fetch occur
    IF emp_record.department_id = 80 THEN
       DBMS_OUTPUT.PUT_LINE ('Employee ' || emp_record.last_name
                             || ' works in the Sales Dept. ');
    END IF;
  END LOOP;   --implicit close occurs
END ;
/
```

Example

Retrieve the first five employees with a job history.

```
SET SERVEROUTPUT ON
DECLARE
  v_employee_id  employees.employee_id%TYPE;
  v_job_id       employees.job_id%TYPE;
  v_start_date   DATE;
  v_end_date     DATE;
  CURSOR emp_cursor IS
     SELECT    employee_id, job_id, start_date, end_date
     FROM job_history
     ORDER BY   employee_id;
BEGIN
  OPEN emp_cursor;
  LOOP
    FETCH emp_cursor
      INTO v_employee_id, v_job_id, v_start_date, v_end_date;
    DBMS_OUTPUT.PUT_LINE (’Employee #: ’ || v_employee_id ||
      ’ held the job of ’ || v_job_id || ’ FROM ’ ||
      v_start_date || ’ TO ’ || v_end_date);
    EXIT WHEN emp_cursor%ROWCOUNT > 4 OR
      emp_cursor%NOTFOUND;
  END LOOP;
  CLOSE emp_cursor;
END;
/
```

Summary

In this lesson you should have learned to do the following:

- **Distinguish cursor types:**
  - **Implicit cursors: used for all `DML` statements and single-row queries**
  - **Explicit cursors: used for queries of zero, one, or more rows**
- **Manipulate explicit cursors**
- **Evaluate the cursor status by using cursor attributes**
- **Use cursor `FOR` loops**

ORACLE

Summary

Oracle uses work areas to execute SQL statements and store processing information. A PL/SQL construct called a cursor allows you 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 can 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 or cursor variable, these attributes return useful information about the execution of a data manipulation statement. You can use cursor attributes in procedural statements but not in SQL statements.

Practice 6 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**

ORACLE

Practice 6 Overview

This practice applies 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.

Practice 6

1. Run the command in the script `lab6_1.sql` to create a new table for storing the salaries of the employees.

```
CREATE TABLE    top_dogs
( salary   NUMBER(8,2));
```

2. Create a PL/SQL block that determines the top employees with respect to salaries.
   a. 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 earners, enter 5.
   Note: Use the `DEFINE` command to provide the value for *n*. Pass the value to the PL/SQL block through a *i*SQL*Plus substitution variable.
   b. In a loop use the *i*SQL*Plus substitution parameter created in step 1 and gather the salaries of the top *n* people from the `EMPLOYEES` table. There should be no duplication in the salaries. If two employees earn the same salary, the salary should be picked up only once.
   c. Store the salaries in the `TOP_DOGS` table.
   d. 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_DOGS` table after each test. The output shown represents the five highest salaries in the `EMPLOYEES` table.

SALARY
24000
17000
14000
13500
13000

3. Create a PL/SQL block that does the following:
   a. Use the `DEFINE` command to provide the department ID. Pass the value to the PL/SQL block through a *i*SQL*Plus substitution variable.
   b. In a PL/SQL block, retrieve the last name, salary, and `MANAGER ID` of the employees working in that department.
   c. 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`.

Note: `SET ECHO OFF` to avoid displaying the PL/SQL code every time you execute the script.

d. 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 Due for a raise Kaufling Due for a raise Vollman Due for a raise Mourgas 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 . . . . . .

7

Advanced Explicit Cursor Concepts

ORACLE

Objectives

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

- **Write a cursor that uses parameters**
- **Determine when a `FOR UPDATE` clause in a cursor is required**
- **Determine when to use the `WHERE CURRENT OF` clause**
- **Write a cursor that uses a subquery**

ORACLE

Lesson Aim

In this lesson, you learn more about writing explicit cursors, specifically about writing cursors that use parameters.

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,.....) ;
```

ORACLE

Cursors with Parameters

You can pass parameters to the 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 re-opened 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 previously declared cursor.
parameter_name	is the name of a parameter.

`parameter_name`

datatype	is a scalar data type of the parameter.
select_statement	is a `SELECT` statement without the `INTO` clause.

When the cursor is opened, you pass values to each of the parameters by position or by name. You can pass values from PL/SQL or host variables as well as from literals.

Note: 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

Pass the department number and job title to the `WHERE` clause, in the cursor `SELECT` statement.

```
DECLARE
  CURSOR emp_cursor
  (p_deptno NUMBER, p_job VARCHAR2) IS
     SELECT employee_id, last_name
     FROM   employees
     WHERE  department_id = p_deptno
     AND    job_id = p_job;
BEGIN
  OPEN emp_cursor (80, 'SA_REP');
  . . .
  CLOSE emp_cursor;
  OPEN emp_cursor (60, 'IT_PROG');
  . . .
END;
```

ORACLE

Cursors with Parameter

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 cursor's query. In the following example, a cursor is declared and is defined with two parameters.

```
DECLARE
    CURSOR emp_cursor(p_deptno NUMBER, p_job VARCHAR2) IS
    SELECT ...
```

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

```
OPEN emp_cursor(60, v_emp_job);
OPEN emp_cursor(90, 'AD_VP');
```

You can pass parameters to the cursor 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(50, 'ST_CLERK') LOOP ...
```

The FOR UPDATE Clause

Syntax:

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

- **Explicit locking allows you to deny access for the duration of a transaction.**
- **Lock the rows *before* the update or delete.**

ORACLE

The FOR UPDATE Clause

You may want to lock rows before you update or delete rows. Add the FOR UPDATE clause in the cursor query to lock the affected rows when the cursor is opened. Because the Oracle Server releases locks at the end of the transaction, you should not commit across fetches from an explicit cursor if FOR UPDATE is used.

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 error if the rows are locked by another session

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

The SELECT ... FOR UPDATE statement identifies the rows that will be updated or deleted, 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 the row is not changed by another user before the update.

The optional NOWAIT keyword tells Oracle 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 , Oracle waits until the rows are available.

The FOR UPDATE Clause

Retrieve the employees who work in department 80 and update their salary.

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

ORACLE

The FOR UPDATE Clause (continued)

Note: If the Oracle server cannot acquire the locks on the rows it needs in a SELECT FOR UPDATE, it waits indefinitely. You can use the NOWAIT clause in the SELECT FOR UPDATE statement and test for the error code that returns because of failure to acquire the locks in a loop. You can retry opening the cursor *n* times before terminating the PL/SQL block. If you have a large table, you can achieve better performance by using the LOCK TABLE statement to lock all rows in the table. However, when using LOCK TABLE, you cannot use the WHERE CURRENT OF clause and must use the notation WHERE *column = identifier*.

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

Note: The WHERE CURRENT OF clause is explained later in this lesson.

The FOR UPDATE clause identifies the rows that will be updated or deleted, 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 the row is not changed by another user before the update.

The 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.**

ORACLE

The WHERE CURRENT OF Clause

When referencing the current row from an explicit cursor, use the WHERE CURRENT OF clause. This allows you to apply updates and deletes to the row currently being addressed, without the need to explicitly reference the ROWID. 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.)

The WHERE CURRENT OF Clause

```
DECLARE
CURSOR sal_cursor IS
 SELECT e.department_id, employee_id, last_name, salary
 FROM   employees e, departments d
 WHERE  d.department_id = e.department_id
  and   d.department_id = 60
 FOR UPDATE OF salary NOWAIT;
 BEGIN
 FOR emp_record IN sal_cursor
LOOP
  IF emp_record.salary < 5000 THEN
       UPDATE employees
       SET    salary = emp_record.salary * 1.10
       WHERE CURRENT OF sal_cursor;
  END IF;
END LOOP;
END;
/
```

ORACLE

The WHERE CURRENT OF Clause (continued)

Example

The slide example loops through each employee in department 60, and checks whether the salary is less than 5000. If the salary is less than 5000, the salary is raised by 10%. The WHERE CURRENT OF clause in the UPDATE statement refers to the currently fetched record. Observe that a table can be updated with the WHERE CURRENT OF clause, even if there is a join in the cursor declaration.

Additionally, you can write a DELETE or UPDATE statement to contain the WHERE CURRENT OF *cursor_name* clause to refer to the latest row processed by the FETCH statement. You can update rows based on criteria from a cursor. When you use this clause, the cursor you reference must exist and must contain the FOR UPDATE clause in the cursor query; otherwise, you will receive an error. This clause allows you to apply updates and deletes to the currently addressed row without the need to explicitly reference the ROWID pseudo column.

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;
...
```

ORACLE

Cursors with Subqueries

A subquery is a query (usually enclosed by parentheses) that appears within another SQL data manipulation 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 this example, the subquery creates a data source consisting of department numbers and employee head count in each department (known as 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 will contain the department number, department name, and total number of employees working for the department, provided there are three or more employees working for the department.

Summary

In this lesson, you should have learned how to do the following:

- **Return different active sets using cursors with parameters.**
- **Define cursors with subqueries and correlated subqueries.**
- **Manipulate explicit cursors with commands using the:**
  - **`FOR UPDATE` clause**
  - **`WHERE CURRENT OF` clause**

ORACLE

Summary

An explicit cursor can take parameters. In a query, you can specify a cursor parameter wherever a constant appears. An advantage of using parameters is that you can decide the active set at run time.

PL/SQL provides a method to modify the rows that have been retrieved by the cursor. The method consists of two parts. The `FOR UPDATE` clause in the cursor declaration and the `WHERE CURRENT OF` clause in an `UPDATE` or `DELETE` statement.

Practice 7 Overview

This practice covers the following topics:

- **Declaring and using explicit cursors with parameters**
- **Using a `FOR UPDATE` cursor**

ORACLE

Practice 7 Overview

This practice applies your knowledge of cursors with parameters to process a number of rows from multiple tables.

Practice 7

1. In a loop, use a cursor to retrieve the department number and the department name from the `DEPARTMENTS` table for those departments whose `DEPARTMENT_ID` is less than 100. Pass the department number to another cursor 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.

```
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)

2. Modify the code in `sol4_4.sql` to incorporate a cursor using the `FOR UPDATE` and `WHERE CURRENT OF` functionality in cursor processing.

```
DEFINE p_empno=104
DEFINE p_empno=174
DEFINE p_empno=176
```

EMPLOYEE_ID	SALARY	STARS
104	6000	******
174	11000	**********
176	8600	*********

8 Handling Exceptions

Objectives

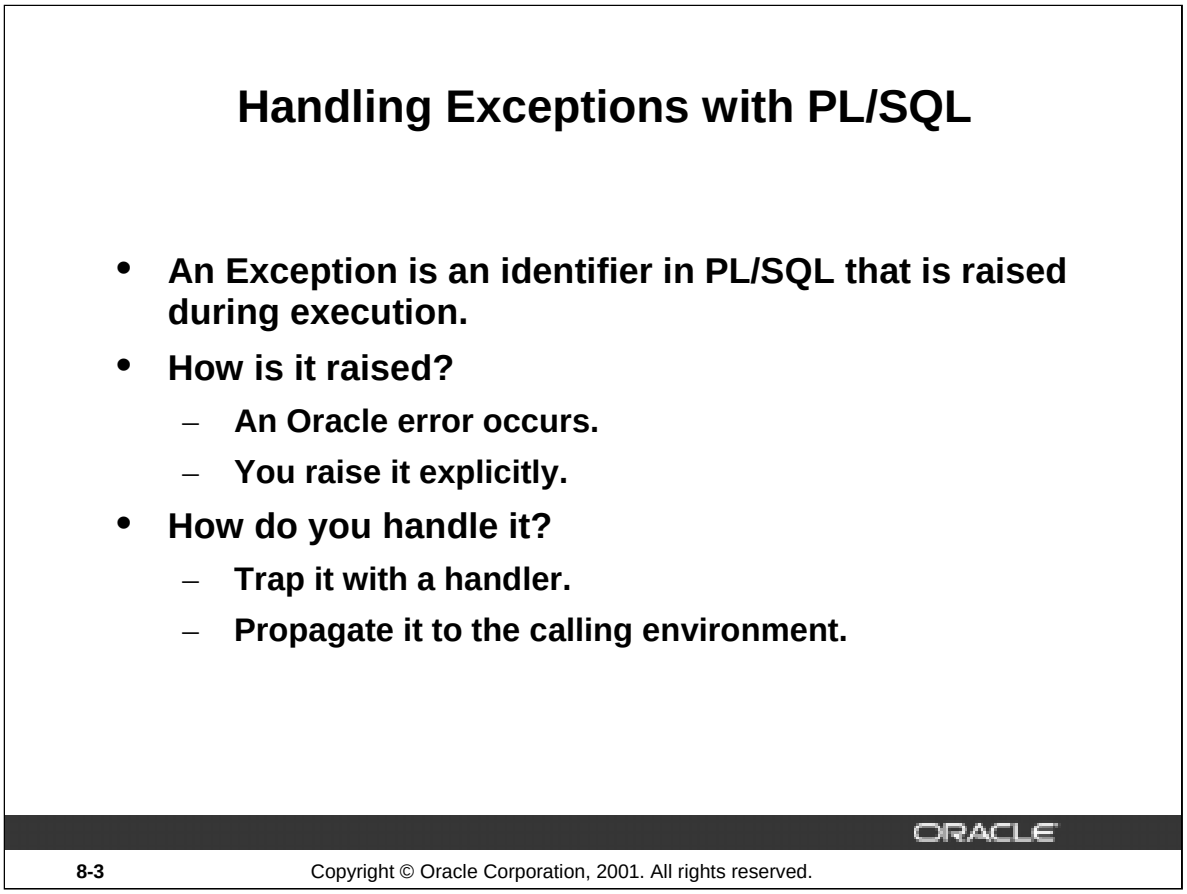
After completing this lesson, you should be able to do the following:

- **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**

ORACLE

Lesson Aim

In this lesson, you learn what PL/SQL exceptions are and how to deal with them using predefined, nonpredefined, and user-defined exception handlers.



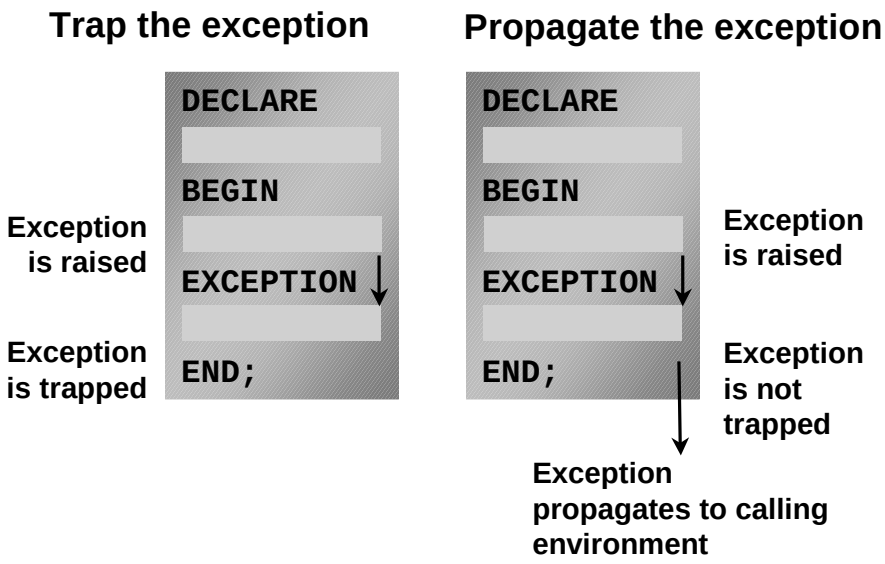
Overview

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

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, then PL/SQL raises the exception `NO_DATA_FOUND`.
- You raise an exception explicitly by issuing the `RAISE` statement within the block. The exception being raised may be either user-defined or predefined.

Handling Exceptions



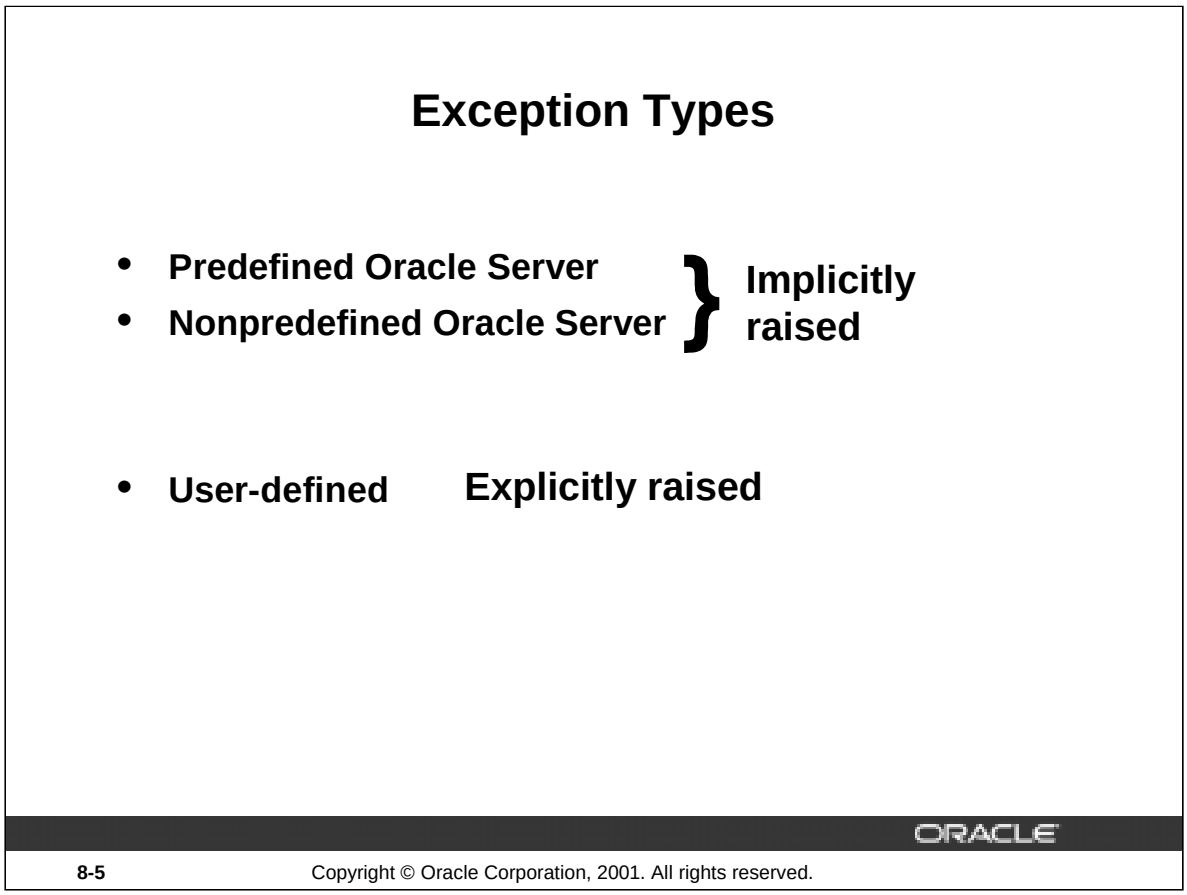
ORACLE

Trapping an Exception

If the exception is raised in the executable section of the block, processing branches to the corresponding exception handler in the exception section of the block. If PL/SQL successfully handles the exception, then the exception does not propagate to the enclosing block or 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 the calling environment.

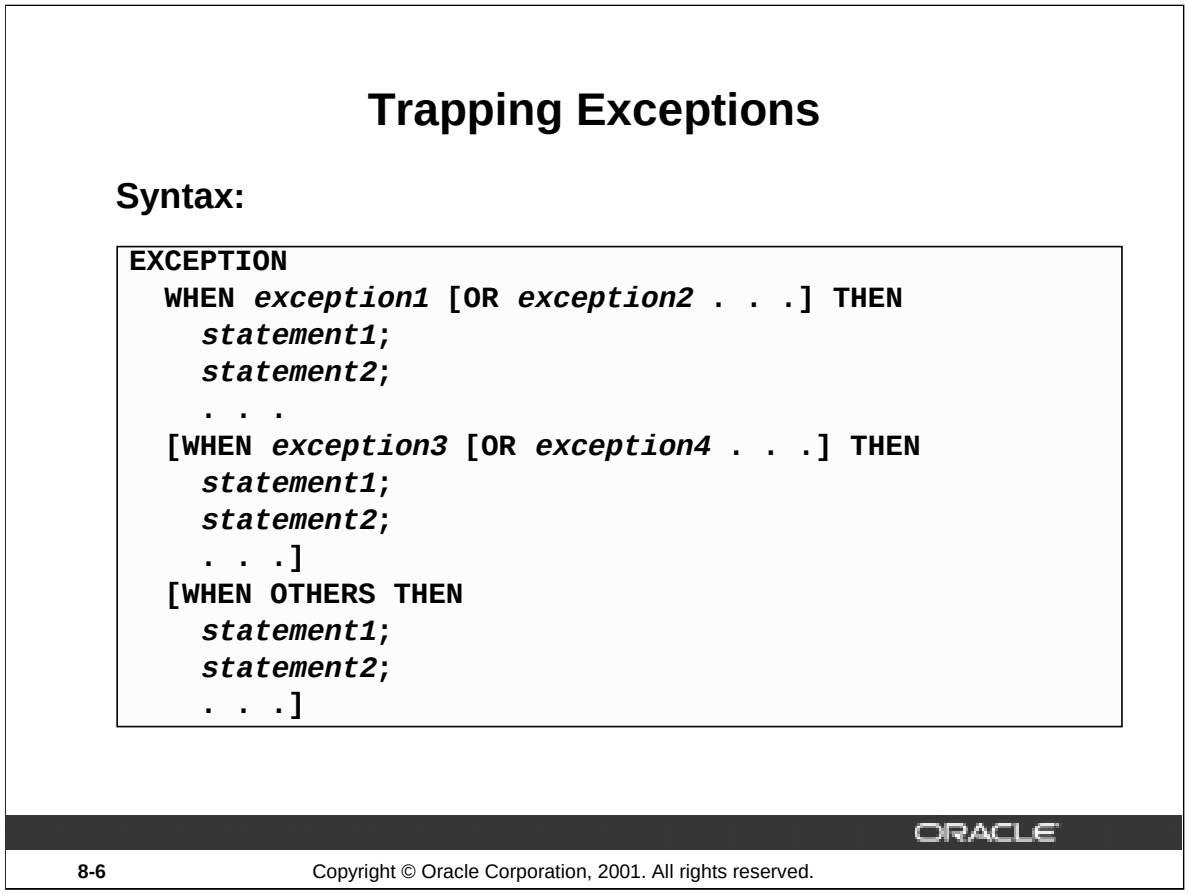


Exception Types

You can program for exceptions to avoid disruption at run time. 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	Do not declare and allow the Oracle server to raise them implicitly
Nonpredefined Oracle Server error	Any other standard Oracle Server error	Declare within the declarative section and allow the Oracle Server to raise them implicitly
User-defined error	A condition that the developer determines is abnormal	Declare within 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

You can trap any error by including a corresponding routine within the exception handling section of the PL/SQL block. Each handler consists of a WHEN clause, which specifies an exception, followed by a sequence of statements to be executed when that exception is raised.

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 unspecified exceptions.

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 is the last exception handler that is defined.

The OTHERS handler traps *all* exceptions 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.

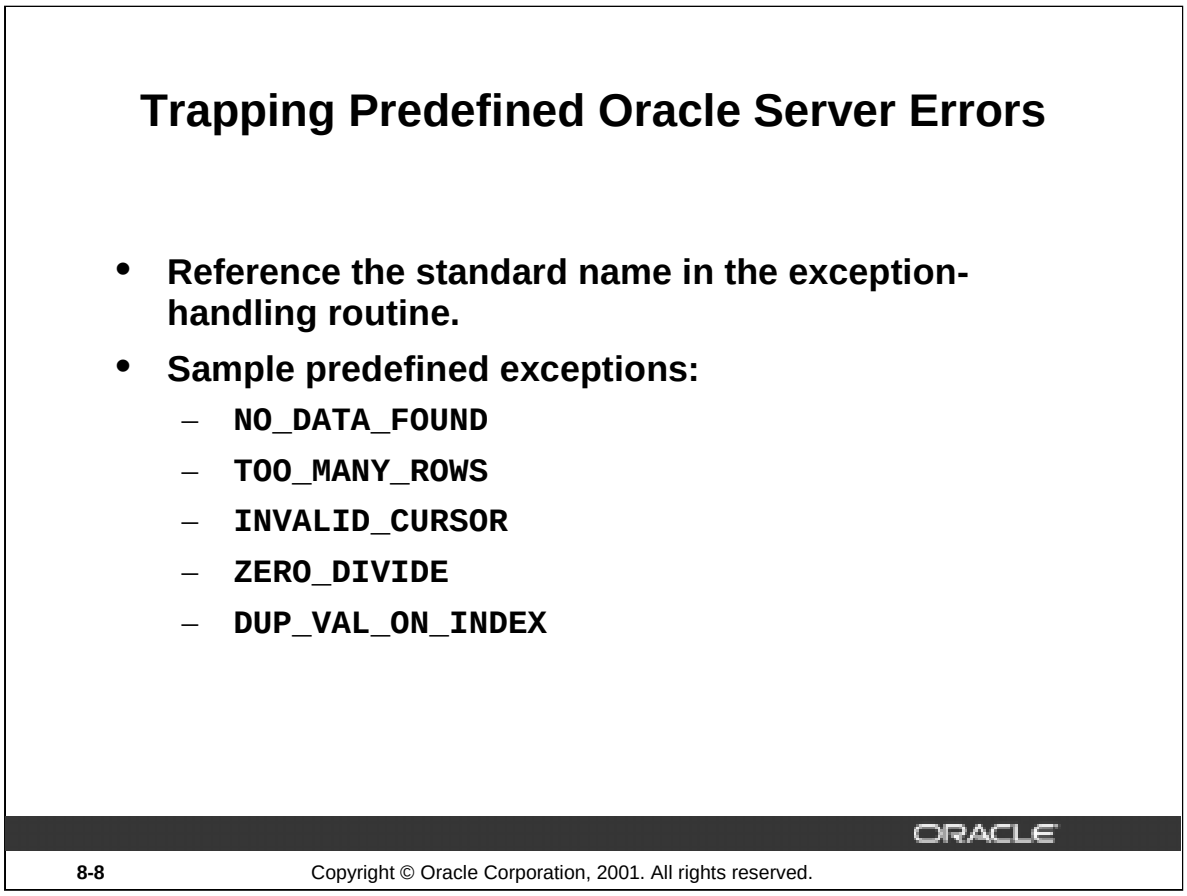
Trapping Exceptions Guidelines

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

ORACLE

Guidelines

- 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

Trap a predefined Oracle Server error by referencing its standard name within the corresponding exception-handling routine.

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

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

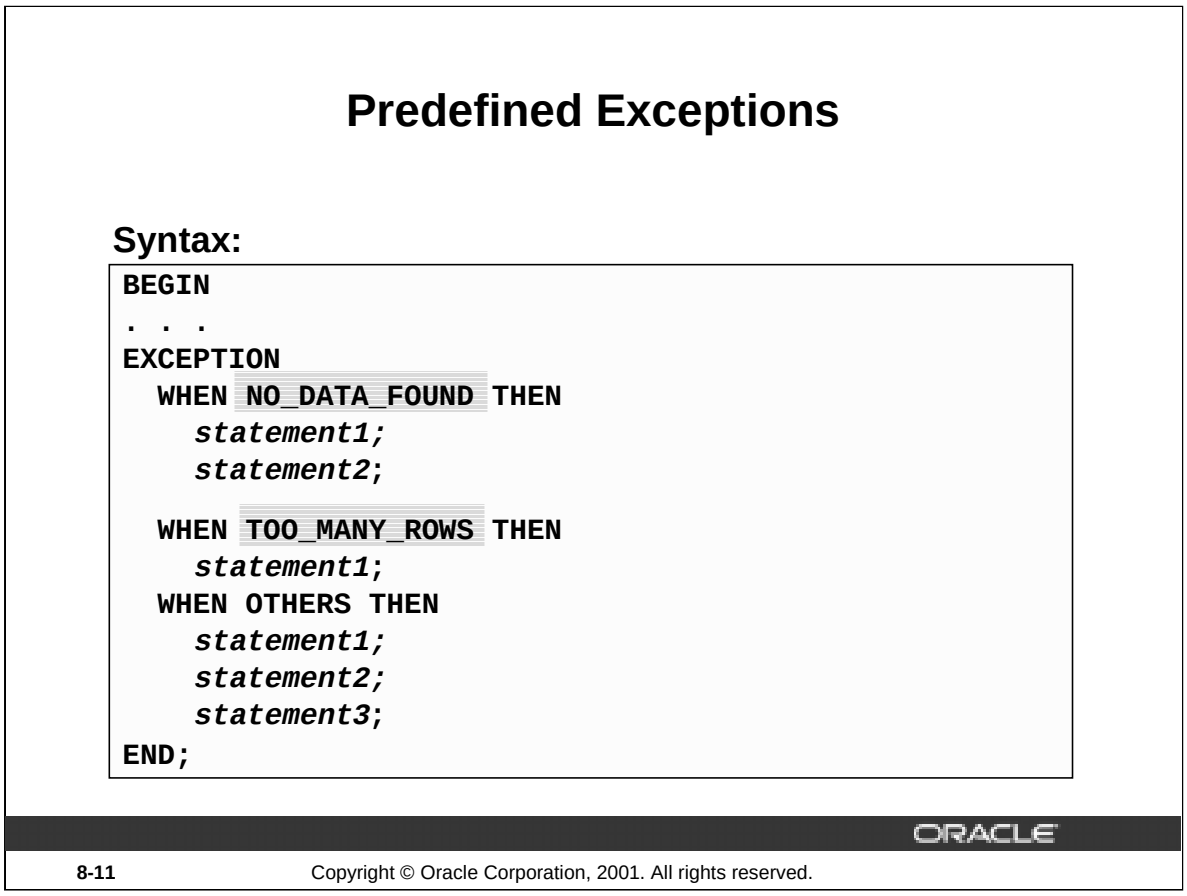
It is a good idea to always handle the `NO_DATA_FOUND` and `TOO_MANY_ROWS` exceptions, which are the most common.

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 is selected, and there is no ELSE clause.
COLLECTION_IS_NULL	ORA-06531	Attempted to apply collection methods other than EXISTS to an uninitialized nested table or varray
CURSOR_ALREADY_OPEN	ORA-06511	Attempted to open an already open cursor
DUP_VAL_ON_INDEX	ORA-00001	Attempted to insert a duplicate value
INVALID_CURSOR	ORA-01001	Illegal cursor operation occurred
INVALID_NUMBER	ORA-01722	Conversion of character string to number fails
LOGIN_DENIED	ORA-01017	Logging on to Oracle 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 Oracle
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 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 using an index number that is outside the legal range (–1 for example)
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 Oracle is 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



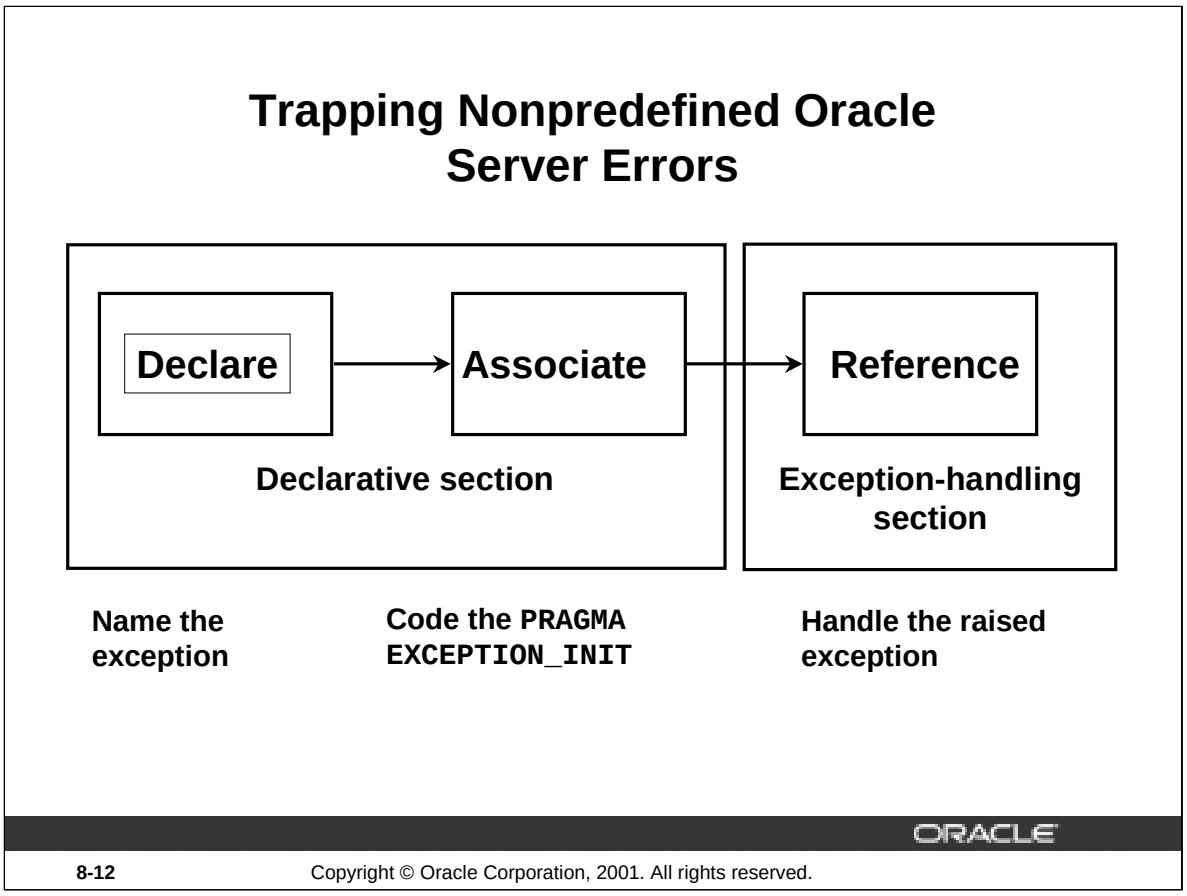
Trapping Predefined Oracle Server Exceptions

When an exception is raised, normal execution of your PL/SQL block or subprogram stops and control transfers to its exception-handling part, which is formatted as shown on the slide.

To catch raised exceptions, you write exception handlers. Each handler consists of a `WHEN` clause, which specifies an exception, followed by a sequence of statements to be executed when that exception is raised. These statements complete execution of the block or subprogram; control does not return to where the exception was raised. In other words, you cannot resume processing where you left off.

The optional `OTHERS` exception handler, which, if present, is always the last handler in a block or subprogram, acts as the handler for all exceptions that are not named specifically. Thus, a block or subprogram can have only one `OTHERS` handler. As the following example shows, use of the `OTHERS` handler guarantees that no exception will go unhandled:

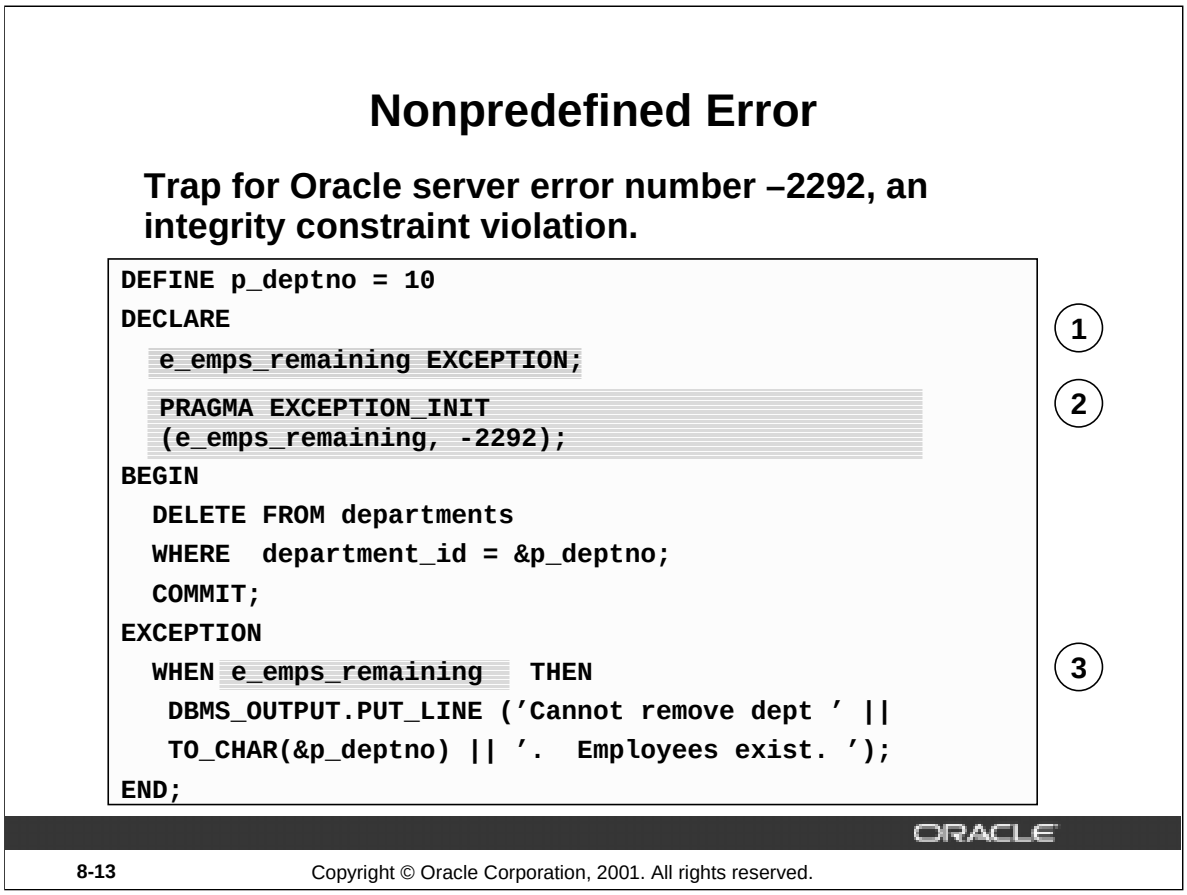
```
EXCEPTION
   WHEN ... THEN
      -- handle the error
   WHEN ... THEN
      -- handle the error
   WHEN OTHERS THEN
      -- handle all other errors
END;
```



Trapping Nonpredefined Oracle Server Errors

You trap a nonpredefined Oracle server error by declaring it first, or by using the `OTHERS` handler. The declared exception is raised implicitly. In PL/SQL, the `PRAGMA EXCEPTION_INIT` tells the compiler to associate an exception name with an Oracle error number. That allows you to refer to any internal exception by name and to write a specific handler for it.

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



Trapping a Nonpredefined Oracle Server Exception

1. Declare the name for the exception within the declarative section.

   Syntax

   ```
   exception     EXCEPTION;
   ```

   where: *exception* is the name of the exception.

2. Associate the declared exception with the standard Oracle server error number using the PRAGMA EXCEPTION_INIT statement.

   Syntax

   ```
   PRAGMA        EXCEPTION_INIT(exception, error_number);
   ```

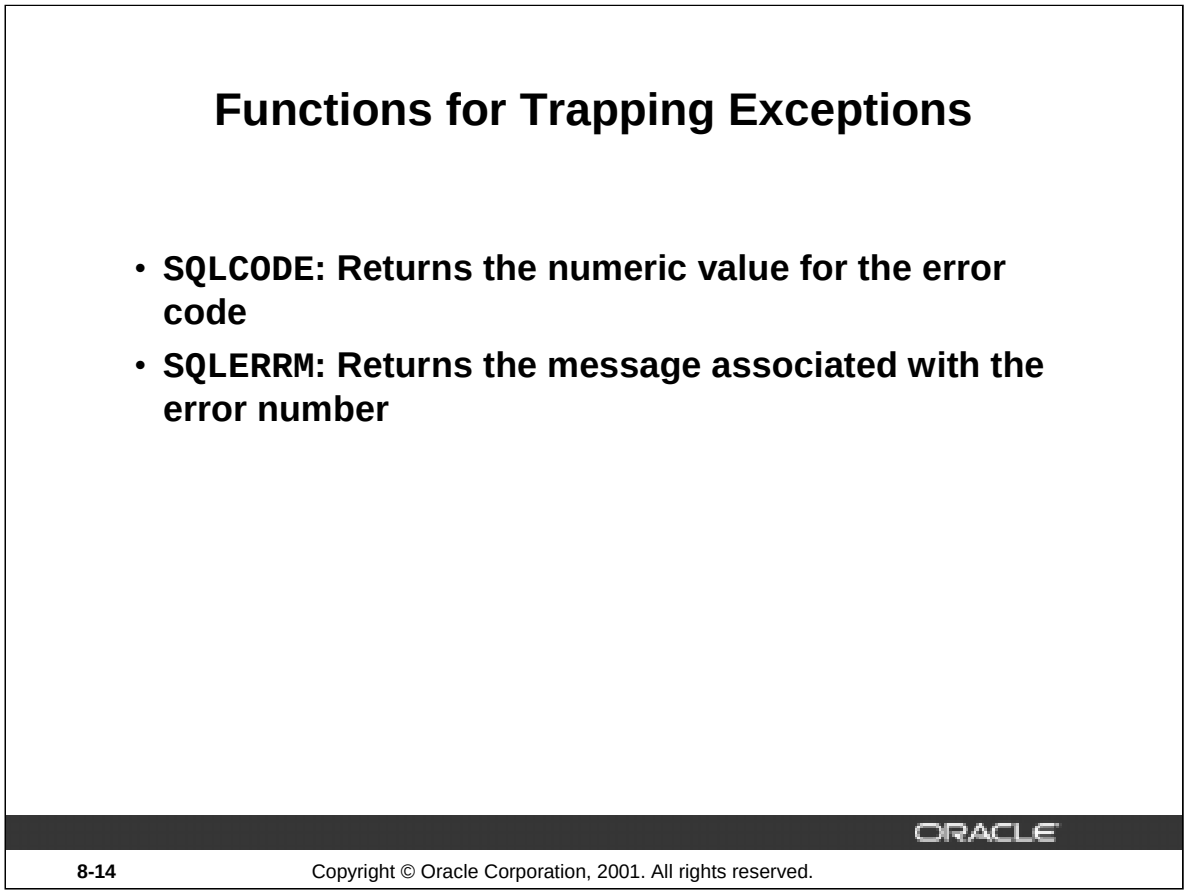
   where: *exception* is the previously declared exception.

   error_number is a standard Oracle Server error number.

3. Reference the declared exception within the corresponding exception-handling routine.

Example

If there are employees in a department, print a message to the user that the department cannot be removed.



Error-Trapping Functions

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 message, you can decide which subsequent action to take based on the error.

`SQLCODE` returns the number of the Oracle error for internal exceptions. You can pass an error number to `SQLERRM`, which then 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

Example `SQLCODE` Values

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
  v_error_code      NUMBER;
  v_error_message   VARCHAR2(255);
BEGIN
...
EXCEPTION
...
  WHEN OTHERS THEN
    ROLLBACK;
    v_error_code := SQLCODE ;
    v_error_message := SQLERRM ;
    INSERT INTO errors
    VALUES(v_error_code, v_error_message);
END;
```

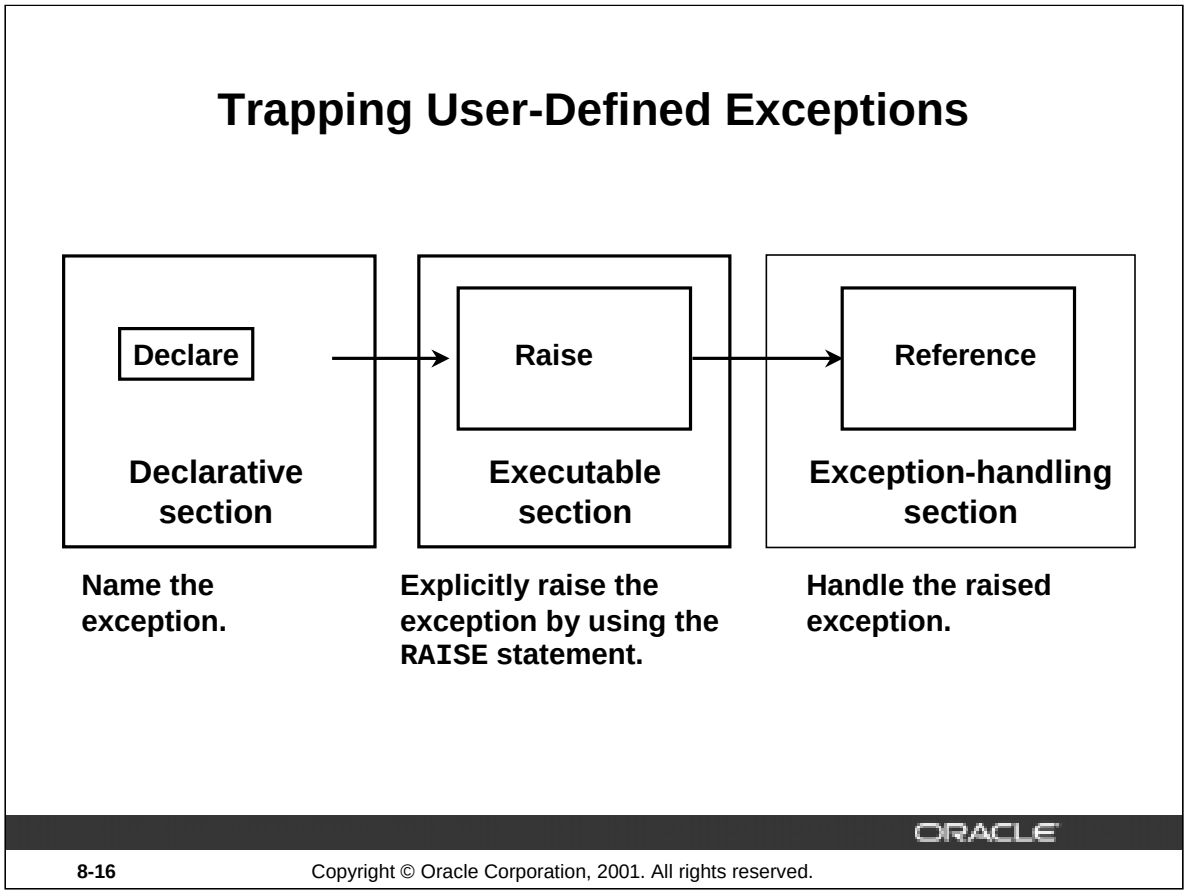

ORACLE

Error-Trapping Functions (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 on the slide illustrates the values of `SQLCODE` and `SQLERRM` being 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, then use the variables in the SQL statement, as shown in the following example:

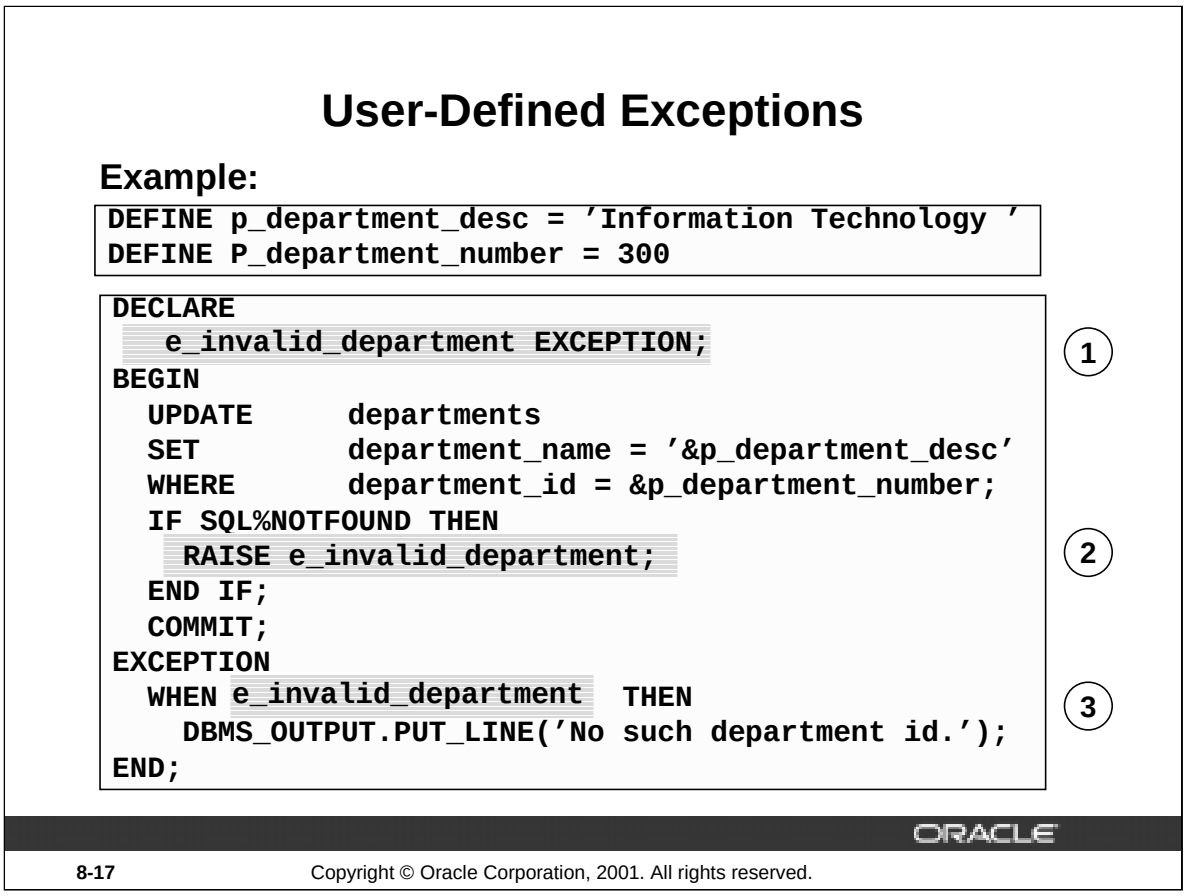
```
DECLARE
   err_num NUMBER;
   err_msg VARCHAR2(100);
BEGIN
   ...
EXCEPTION
   ...
   WHEN OTHERS THEN
      err_num := SQLCODE;
      err_msg := SUBSTR(SQLERRM, 1, 100);
      INSERT INTO errors VALUES (err_num, err_msg);
END;
```



Trapping User-Defined Exceptions

PL/SQL allows you to define your own exceptions. User-defined PL/SQL exceptions must be:

- Declared in the declare section of a PL/SQL block
- Raised explicitly with `RAISE` statements



Trapping User-Defined Exceptions (continued)

You trap a user-defined exception by declaring it and raising it explicitly.

1. Declare the name for the user-defined exception within the declarative section.

   Syntax:

   exception EXCEPTION;

   where: *exception* is the name of the exception

2. Use the RAISE statement to raise the exception explicitly within the executable section.

   Syntax:

   RAISE *exception*;

   where: *exception* is the previously declared exception

3. Reference the declared exception within the corresponding exception-handling routine.

Example

This block updates the description 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 will be 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 back to the calling environment.

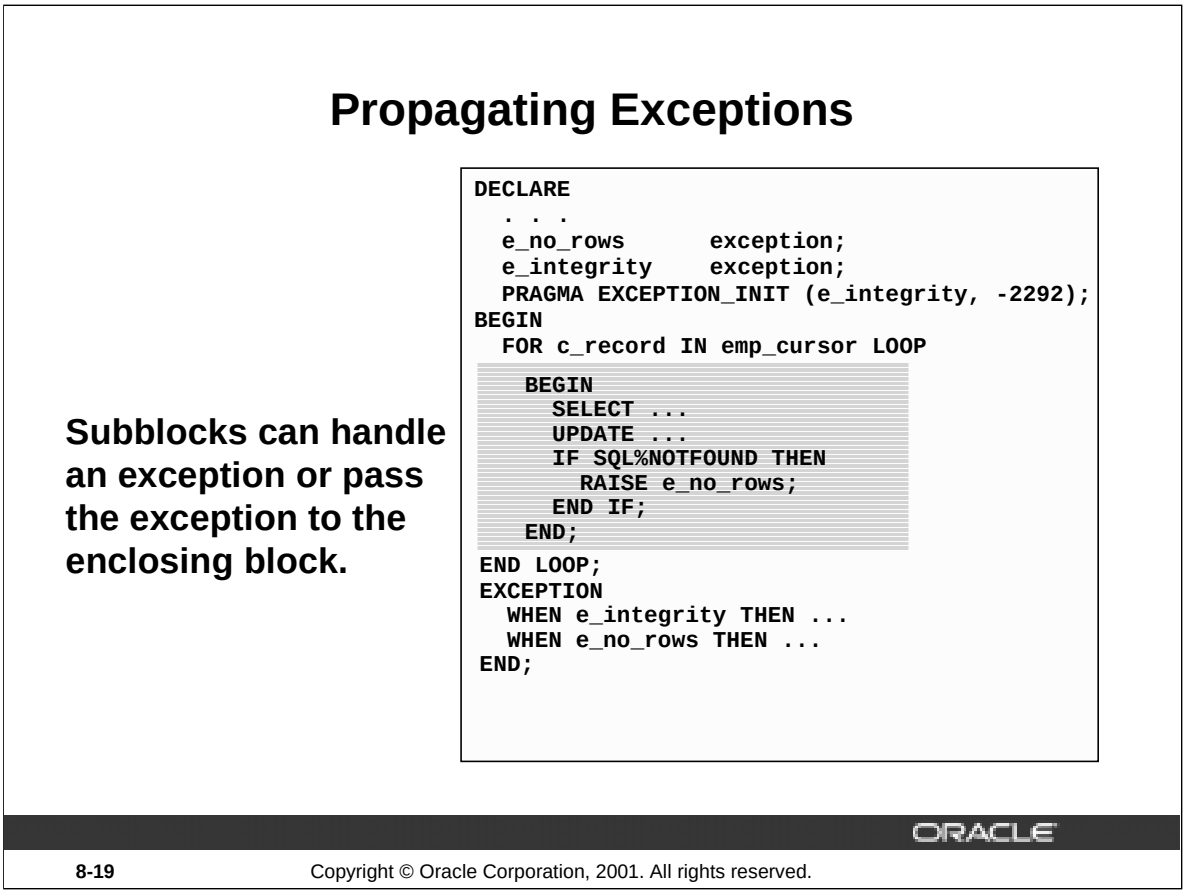
Calling Environments

*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 a 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

ORACLE

Propagating Exceptions

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 an Exception in a Subblock

When a subblock handles an exception, it terminates normally, and control resumes in the enclosing block immediately after the subblock END statement.

However, if PL/SQL raises an exception and the current block does not have a handler for that exception, the exception propagates in 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.

Observe in the example that the `exceptions`, `e_no_rows` and `e_integrity`, are declared in the outer block. In the inner block, when the `e_no_rows` exception is raised, PL/SQL looks for the exception in the sub block. Because the exception is not declared in the subblock, the exception propagates to the outer block, where PL/SQL finds the declaration.

The RAISE_APPLICATION_ERROR Procedure

Syntax:

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

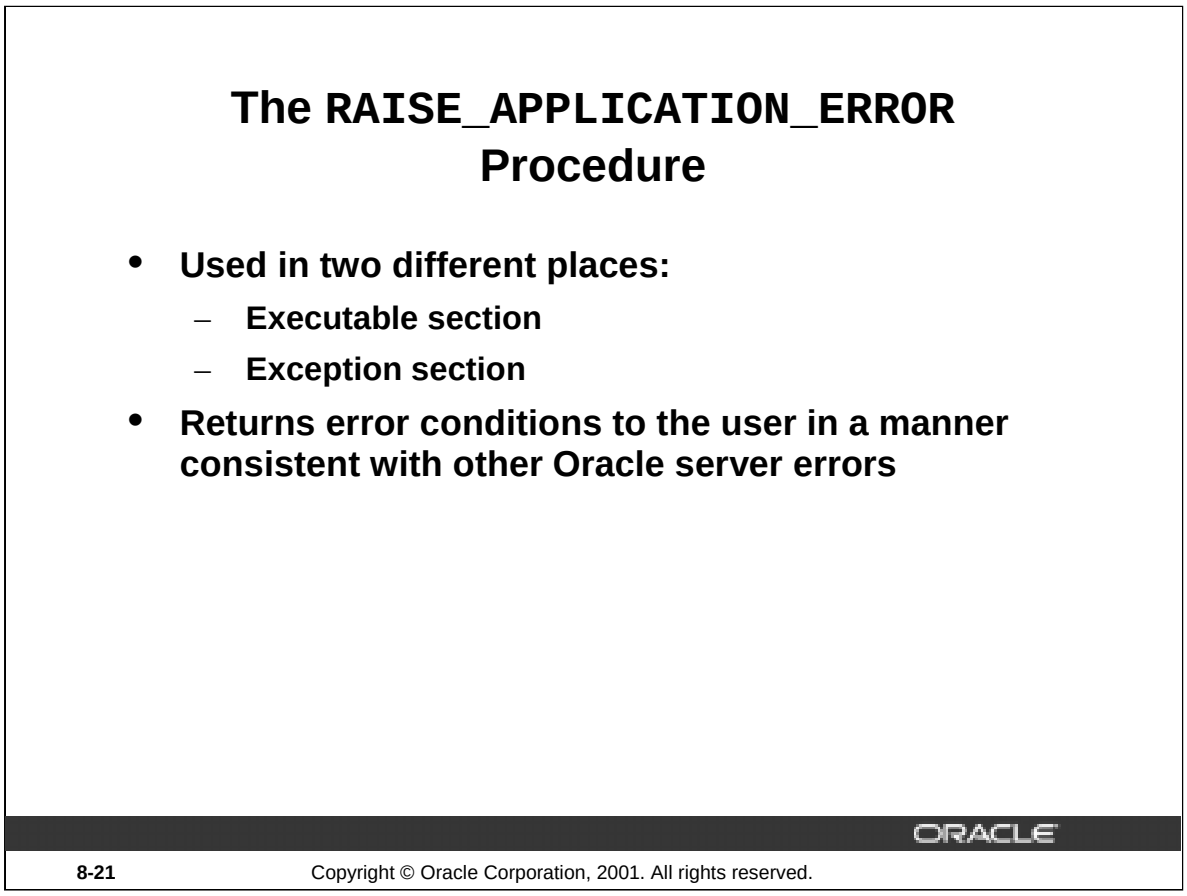
- **You can use this procedure to issue user-defined error messages from stored subprograms.**
- **You can report errors to your application and avoid returning unhandled exceptions.**

The RAISE_APPLICATION_ERROR Procedure

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

In the syntax:

error_number	is a user-specified number for the exception between –20000 and –20999.
message	is the user-specified message for the exception. It is a character string up to 2,048 bytes long.
TRUE | FALSE	is an optional Boolean parameter (If TRUE, the error is placed on the stack of previous errors. If FALSE, the default, the error replaces all previous errors.)



The RAISE_APPLICATION_ERROR Procedure (continued)

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

RAISE_APPLICATION_ERROR

Executable section:

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

Exception section:

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

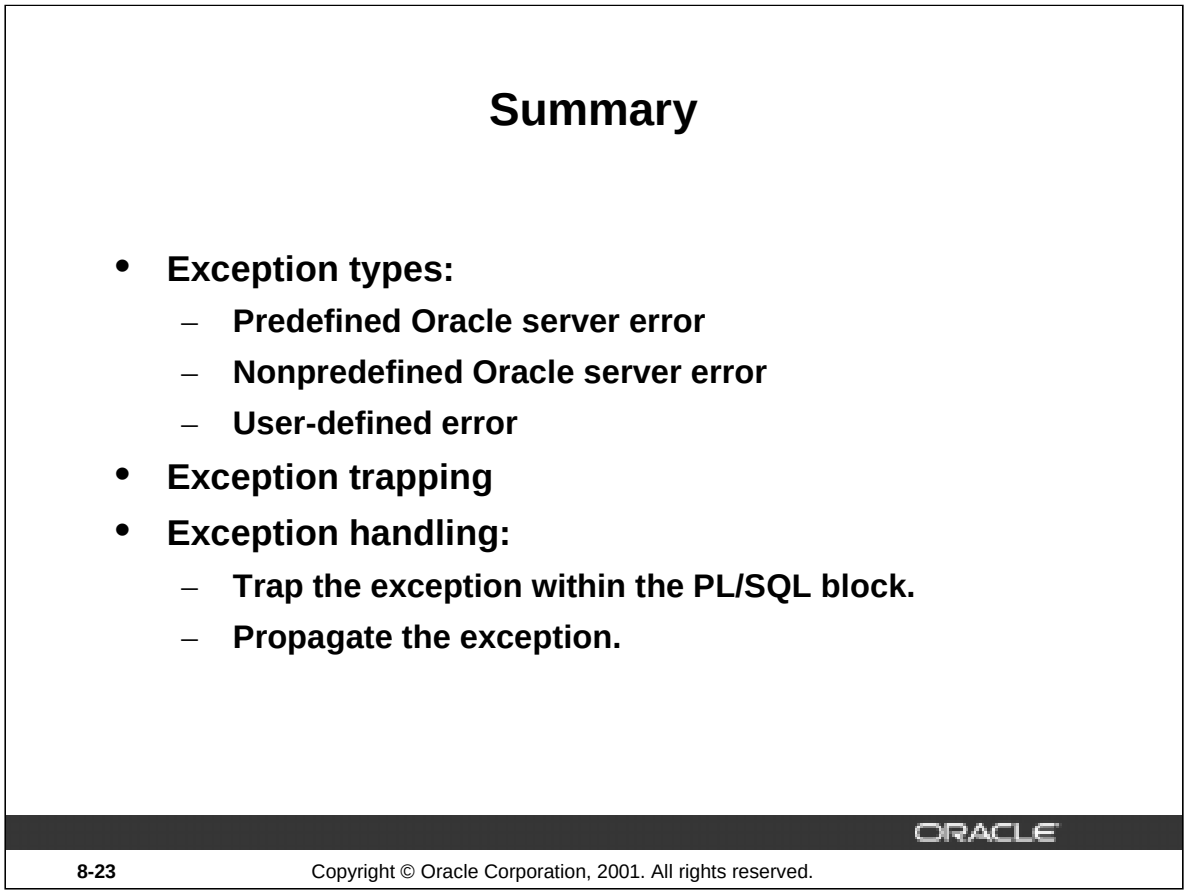
ORACLE

Example

The slide shows that the RAISE_APPLICATION_ERROR procedure can be used in both the executable and exception sections of a PL/SQL program.

Here is another example of RAISE_APPLICATION_ERROR procedure that can be used in both the executable and exception sections of a PL/SQL program:

```
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 PL/SQL, a warning or error condition is called an exception. Predefined exceptions are error conditions that are defined by the Oracle server. Nonpredefined exceptions are any other standard Oracle Server Error. User-defined exceptions are exceptions specific to your application. Examples of predefined exceptions include division by zero (`ZERO_DIVIDE`) and out of memory (`STORAGE_ERROR`). Exceptions without defined names can be assigned names, using the `PRAGMA EXCEPTION_INIT` statement.

You can define exceptions of your own in the declarative part of any PL/SQL block, subprogram, or package. For example, you can define an exception named `INSUFFICIENT_FUNDS` to flag overdrawn bank accounts. User-defined exceptions must be given names.

When an error occurs, an exception is raised. That is, normal execution stops and control transfers to the exception-handling part of your PL/SQL block or subprogram. Internal exceptions are raised implicitly (automatically) by the run-time system. User-defined exceptions must be raised explicitly by `RAISE` statements, which can also raise predefined exceptions.

To handle raised exceptions, you write separate routines called exception handlers. After an exception handler runs, the current block stops executing and the enclosing block resumes with the next statement. If there is no enclosing block, control returns to the host environment.

Practice 8 Overview

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

ORACLE

Practice 8 Overview

In this practice, you create exception handlers for specific situations.

Practice 8

1. Write a PL/SQL block to select the name of the employee with a given salary value.
   a. Use the `DEFINE` command to provide the salary. Pass the value to the PL/SQL block through a *i*SQL*Plus substitution variable. 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*>."
   b. 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*>."
   c. If the salary entered returns only one row, insert into the `MESSAGES` table the employee's name and the salary amount.
   d. Handle any other exception with an appropriate exception handler and insert into the `MESSAGES` table the message "Some other error occurred."
   e. Test the block for a variety of test cases. Display the rows from the `MESSAGES` table to check whether the PL/SQL block has executed successfully. Some sample output is shown below.

No employee with a salary of 5000
More than one employee with a salary of 6000
More than one employee with a salary of 7000
No employee with a salary of 2000

2. Modify the code in `p3q3.sql` to add an exception handler.
   a. Use the `DEFINE` command to provide the department ID and department location. Pass the values to the PL/SQL block through a *i*SQL*Plus substitution variables.
   b. Write an exception handler for the error to pass a message to the user that the specified department does not exist. Use a bind variable to pass the message to the user.
   c. Execute the PL/SQL block by entering a department that does not exist.

G_MESSAGE
Department 200 is an invalid department

Practice 8 (continued)

3. Write a PL/SQL block that prints the number of employees who earn plus or minus $100 of the salary value set for an *i*SQL*Plus substitution variable. Use the `DEFINE` command to provide the salary value. Pass the value to the PL/SQL block through a *i*SQL*Plus substitution variable.
   a. If there is no employee within that salary range, print a message to the user indicating that is the case. Use an exception for this case.
   b. If there are one or more employees within that range, the message should indicate how many employees are in that salary range.
   c. Handle any other exception with an appropriate exception handler. The message should Indicate that some other error occurred.

```
DEFINE p_sal = 7000
DEFINE p_sal = 2500
DEFINE p_sal = 6500
```

G_MESSAGE
There is/are 4 employee(s) with a salary between 6900 and 7100

G_MESSAGE
There is/are 12 employee(s) with a salary between 2400 and 2600

G_MESSAGE
There is/are 3 employee(s) with a salary between 6400 and 6600

9

Creating Procedures

ORACLE

Objectives

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

- **Distinguish anonymous PL/SQL blocks from named PL/SQL blocks (subprograms)**
- **List the benefits of using subprograms**
- **List the different environments from which subprograms can be invoked**

ORACLE

Lesson Aim

In this lesson, you learn the difference between anonymous PL/SQL blocks and subprograms. You also learn to create, execute, and remove procedures.

Objectives

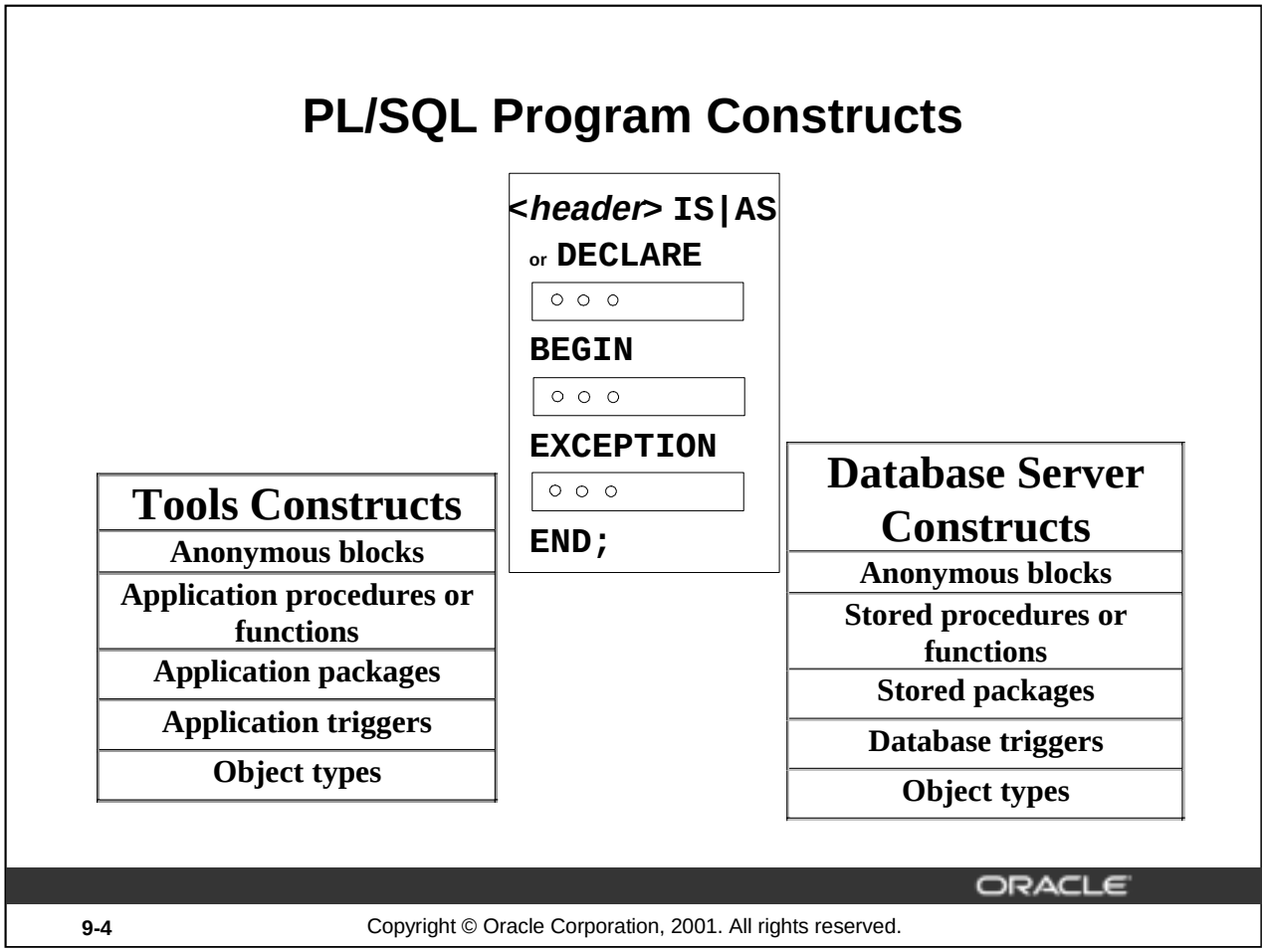
After completing this lesson, you should be able to do the following:

- **Describe PL/SQL blocks and subprograms**
- **Describe the uses of procedures**
- **Create procedures**
- **Differentiate between formal and actual parameters**
- **List the features of different parameter modes**
- **Create procedures with parameters**
- **Invoke a procedure**
- **Handle exceptions in procedures**
- **Remove a procedure**

ORACLE

Lesson Aim

In this lesson, you learn the difference between anonymous PL/SQL blocks and subprograms. You also learn to create, execute, and remove procedures.



PL/SQL Program Constructs

The diagram above displays a variety of different PL/SQL program constructs using the basic PL/SQL block. In general, a block is either an anonymous block or a named block (known as a subprogram or program unit).

PL/SQL Block Structure

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

Note: In the slide, the word "or" prior to the keyword DECLARE is not part of the syntax. It is used in the diagram to differentiate between starting of subprograms and anonymous blocks.

The PL/SQL blocks can be constructed on and use the Oracle server (stored PL/SQL program units). They can also be constructed using the Oracle Developer tools such as Oracle Forms Developer, Oracle Report Developer, and so on (application or client-side PL/SQL program units).

Object types are user-defined composite data types that encapsulates a data structure along with the functions and procedures needed to manipulate the data. You can create object types either on the Oracle server or using the Oracle Developer tools

In this course, you learn to write and manage stored procedures and functions, database triggers, and packages. Creating object types is not covered in this course.

Overview of Subprograms

A subprogram:

- **Is a named PL/SQL block that can accept parameters and be invoked from a calling environment**
- **Is of two types:**
  - **A procedure that performs an action**
  - **A function that computes a value**
- **Is based on standard PL/SQL block structure**
- **Provides modularity, reusability, extensibility, and maintainability**
- **Provides easy maintenance, improved data security and integrity, improved performance, and improved code clarity**

ORACLE

Overview of Subprogram

A subprogram is based on standard PL/SQL structure that contains a declarative section, an executable section, and an optional exception-handling section

A subprogram can be compiled and stored in the database. It provides modularity, extensibility, reusability, and maintainability.

Modularization is the process of breaking up large blocks of code into smaller groups of code called modules. After code is modularized, the modules can be reused by the same program or shared by other programs. It is easier to maintain and debug code of smaller modules than a single large program. Also, the modules can be easily extended for customization by incorporating more functionality, if required, without affecting the remaining modules of the program.

Subprograms provide easy maintenance because the code is located in one place and hence any modifications required to the subprogram can be performed in this single location. Subprograms provide improved data integrity and security. The data objects are accessed through the subprogram and a user can invoke the subprogram only if appropriate access privilege is granted to the user.

Block Structure for Anonymous PL/SQL Blocks

DECLARE (optional)
Declares PL/SQL objects to be used within this block

BEGIN (mandatory)
Defines the executable statements

EXCEPTION (optional)
Defines the actions that take place if an error or exception arises

END; (mandatory)

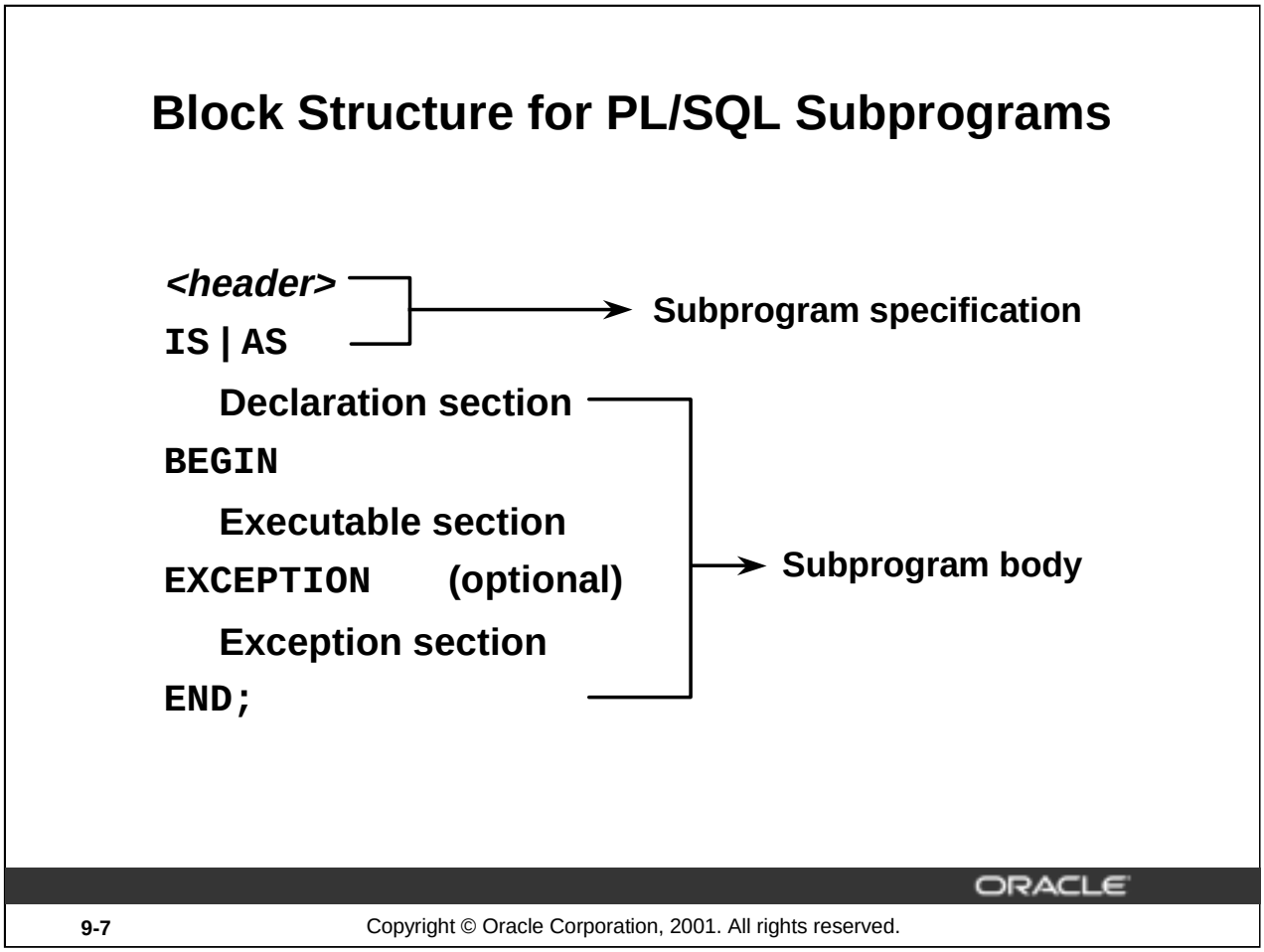
ORACLE

Anonymous Blocks

Anonymous blocks do not have names. You declare them at the point in an application where they are to be run, and they are passed to the PL/SQL engine for execution at run time.

- The section between the keywords `DECLARE` and `BEGIN` is referred to as the declaration section. In the declaration section, you define the PL/SQL objects such as variables, constants, cursors, and user-defined exceptions that you want to reference within the block. The `DECLARE` keyword is optional if you do not declare any PL/SQL objects.
- The `BEGIN` and `END` keywords are mandatory and enclose the body of actions to be performed. This section is referred to as the executable section of the block.
- The section between `EXCEPTION` and `END` is referred to as the exception section. The exception section traps error conditions. In it, you define actions to take if the specified condition arises. The exception section is optional.

The keywords `DECLARE`, `BEGIN`, and `EXCEPTION` are not followed by semicolons, but `END` and all other PL/SQL statements do require semicolons.



Subprograms

Subprograms are named PL/SQL blocks that can accept parameters and be invoked from a calling environment. PL/SQL has two types of subprograms, *procedures* and *functions*.

Subprogram Specification

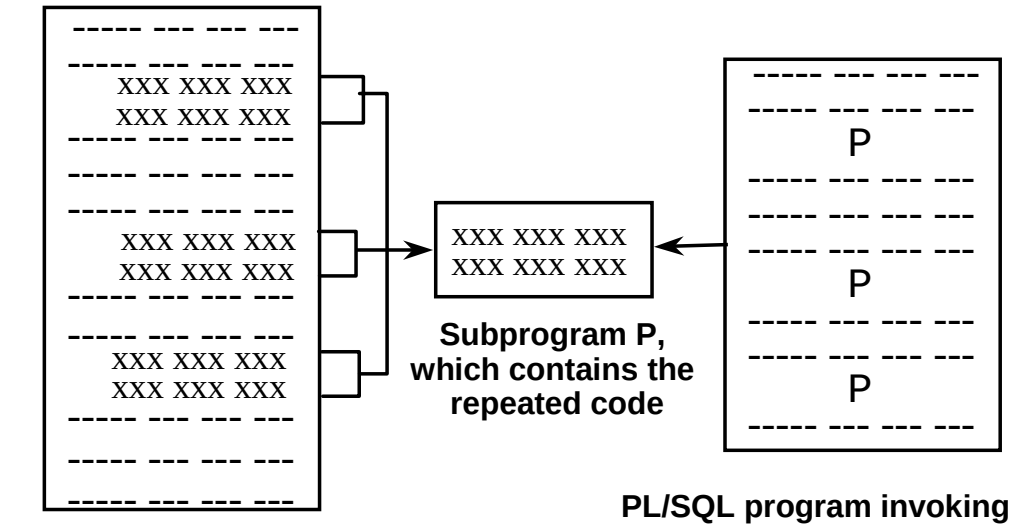
- The header is relevant for named blocks only and determines the way that the program unit is called or invoked.

  The header determines:
  - The PL/SQL subprogram type, that is, either a procedure or a function
  - The name of the subprogram
  - The parameter list, if one exists
  - The `RETURN` clause, which applies only to functions
- The `IS` or `AS` keyword is mandatory.

Subprogram Body

- The declaration section of the block between `IS|AS` and `BEGIN`. The keyword `DECLARE` that is used to indicate the start of the declaration section in anonymous blocks is not used here.
- The executable section between the `BEGIN` and `END` keywords is mandatory, enclosing the body of actions to be performed. There must be at least one statement existing in this section. There should be at least one `NULL;` statement, which is considered an executable statement.
- The exception section between `EXCEPTION` and `END` is optional. This section traps predefined error conditions. In this section, you define actions to take if the specified error condition arises.

PL/SQL Subprograms



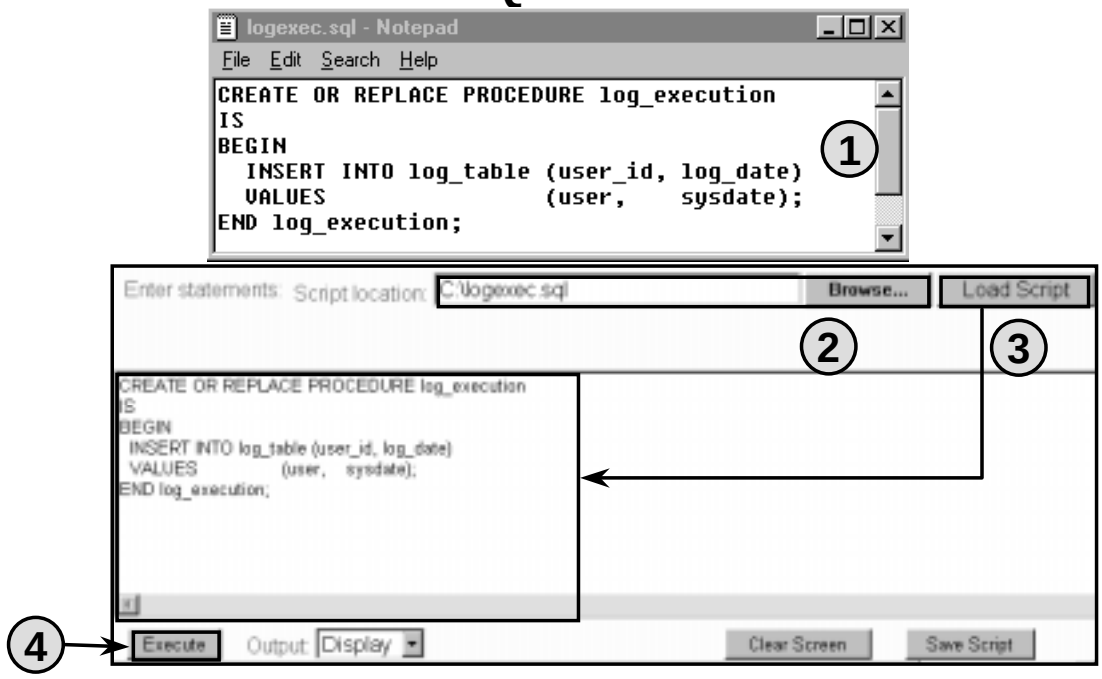
ORACLE

Subprograms

The diagram in the slide explains how you can replace a sequence of PL/SQL statements repeated in a PL/SQL block with a subprogram.

When a sequence of statements is repeated more than once in a PL/SQL subprogram, you can create a subprogram with the repeated code. You can invoke the subprogram at multiple locations in a PL/SQL block. After the subprogram is created and stored in the database, it can be invoked any number of times and from multiple applications.

Developing Subprograms by Using *i*SQL*Plus



Developing Subprograms by Using *i*SQL*Plus

*i*SQL*Plus is an Internet-enabled interface to SQL*Plus. You can use a Web browser to connect to an Oracle database and perform the same actions as you would through other SQL*Plus interfaces.

1. Use a text editor to create a SQL script file to define your subprogram. The example in the slide creates the stored procedure `LOG_EXECUTION` without any parameters. The procedure records the username and current date in a database table called `LOG_TABLE`.

   From *i*SQL*Plus browser window:

2. Use the Browse button to locate the SQL script file.
3. Use the Load Script button to load the script into the iSQL*Plus buffer.
4. Use the Execute button to run the code. By default, the output from the code is displayed on the screen.

PL/SQL subprograms can also be created by using the Oracle development tools such as Oracle Forms Developer.

What Is a Procedure?

- **A procedure is a type of subprogram that performs an action.**
- **A procedure can be stored in the database, as a schema object, for repeated execution.**

ORACLE

Definition of a Procedure

A procedure is a named PL/SQL block that can accept parameters (sometimes referred to as arguments), and be invoked. Generally speaking, you use a procedure to perform an action. A procedure has a header, a declaration section, an executable section, and an optional exception-handling section.

A procedure can be compiled and stored in the database as a schema object.

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

Syntax for Creating Procedures

```
CREATE [OR REPLACE] PROCEDURE procedure_name
 [(parameter1 [mode1] datatype1,
  parameter2 [mode2] datatype2,
  . . .)]
IS|AS
PL/SQL Block;
```

- **The REPLACE option indicates that if the procedure exists, it will be dropped and replaced with the new version created by the statement.**
- **PL/SQL block starts with either BEGIN or the declaration of local variables and ends with either END or END *procedure_name.***

ORACLE

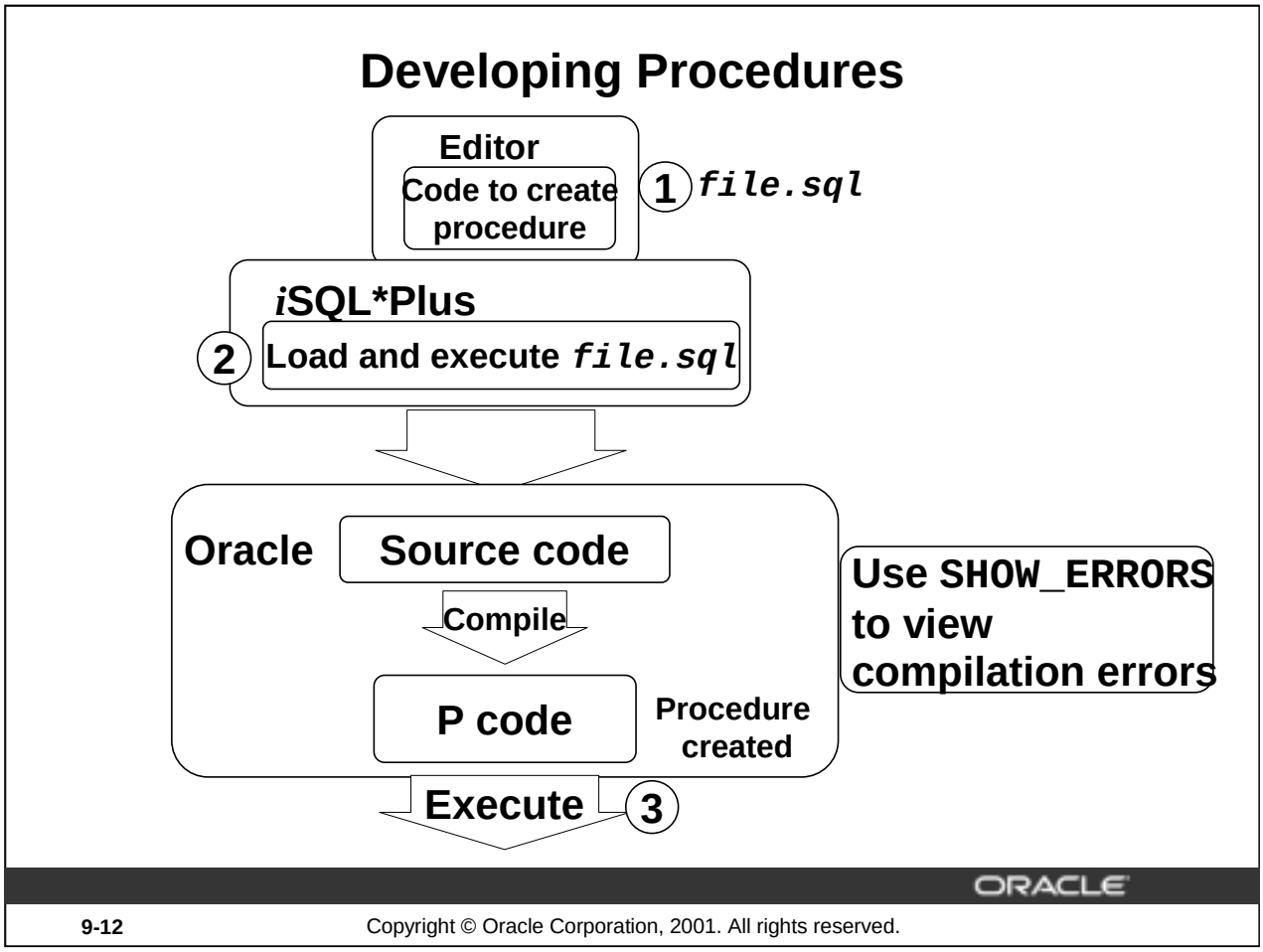
Syntax for Creating Procedures

Syntax Definitions

Parameter	Description
procedure_name	Name of the procedure
parameter	Name of a PL/SQL variable whose value is passed to or populated by the calling environment, or both, depending on the *mode* being used
mode	Type of argument: IN (default) OUT IN OUT
Data type	Data type of the argument–can be any SQL / PLSQL data type. Can be of %TYPE, %ROWTYPE, or any scalar or composite data type.
PL/SQL block	Procedural body that defines the action performed by the procedure

You create new procedures with the CREATE PROCEDURE statement, which may declare a list of parameters and must define the actions to be performed by the standard PL/SQL block. The CREATE clause enables you to create stand-alone procedures, which are stored in an Oracle database.

- PL/SQL blocks start with either BEGIN or the declaration of local variables and end with either END or END *procedure_name*. You cannot reference host or bind variables in the PL/SQL block of a stored procedure.
- The REPLACE option indicates that if the procedure exists, it will be dropped and replaced with the new version created by the statement.
- You can not restrict the size of the data type in the parameters.



Developing Procedures

Following are the main steps for developing a stored procedure. The next two pages provide more detail about creating procedures.

1. Write the syntax: Enter the code to create a procedure (`CREATE PROCEDURE` statement) in a system editor or word processor and save it as a SQL script file (`.sql` extension).
2. Compile the code: Using *i*SQL*Plus, load and run the SQL script file. The source code is compiled into *P code* and the procedure is created.

   A script file with the `CREATE PROCEDURE` (or `CREATE OR REPLACE PROCEDURE`) statement enables you to change the statement if there are any compilation or run-time errors, or to make subsequent changes to the statement. You cannot successfully invoke a procedure that contains any compilation or run-time errors. In *i*SQL*Plus, use `SHOW ERRORS` to see any compilation errors. Running the `CREATE PROCEDURE` statement stores the source code in the data dictionary even if the procedure contains compilation errors.

   Fix the errors in the code using the editor and recompile the code.
3. Execute the procedure to perform the desired action. After the source code is compiled and the procedure is successfully created, the procedure can be executed any number of times using the `EXECUTE` command from *i*SQL*Plus. The PL/SQL compiler generates the *pseudocode* or P code, based on the parsed code. The PL/SQL engine executes this when the procedure is invoked.

Note: If there are any compilation errors, and you make subsequent changes to the `CREATE PROCEDURE` statement, you must either `DROP` the procedure first, or use the `OR REPLACE` syntax.

You can create client side procedures that are used with client side applications using tools such as the Forms and Reports of Oracle IDE. Refer to Appendix C to see how the client side subprograms can be created using the Oracle Procedure Builder tool.

Formal Versus Actual Parameters

- **Formal parameters: variables declared in the parameter list of a subprogram specification**

  Example:

```
CREATE PROCEDURE raise_sal(p_id NUMBER, p_amount NUMBER)
...
END raise_sal;
```

- **Actual parameters: variables or expressions referenced in the parameter list of a subprogram call**

  Example:

```
raise_sal(v_id, 2000)
```

ORACLE

Formal Versus Actual Parameters

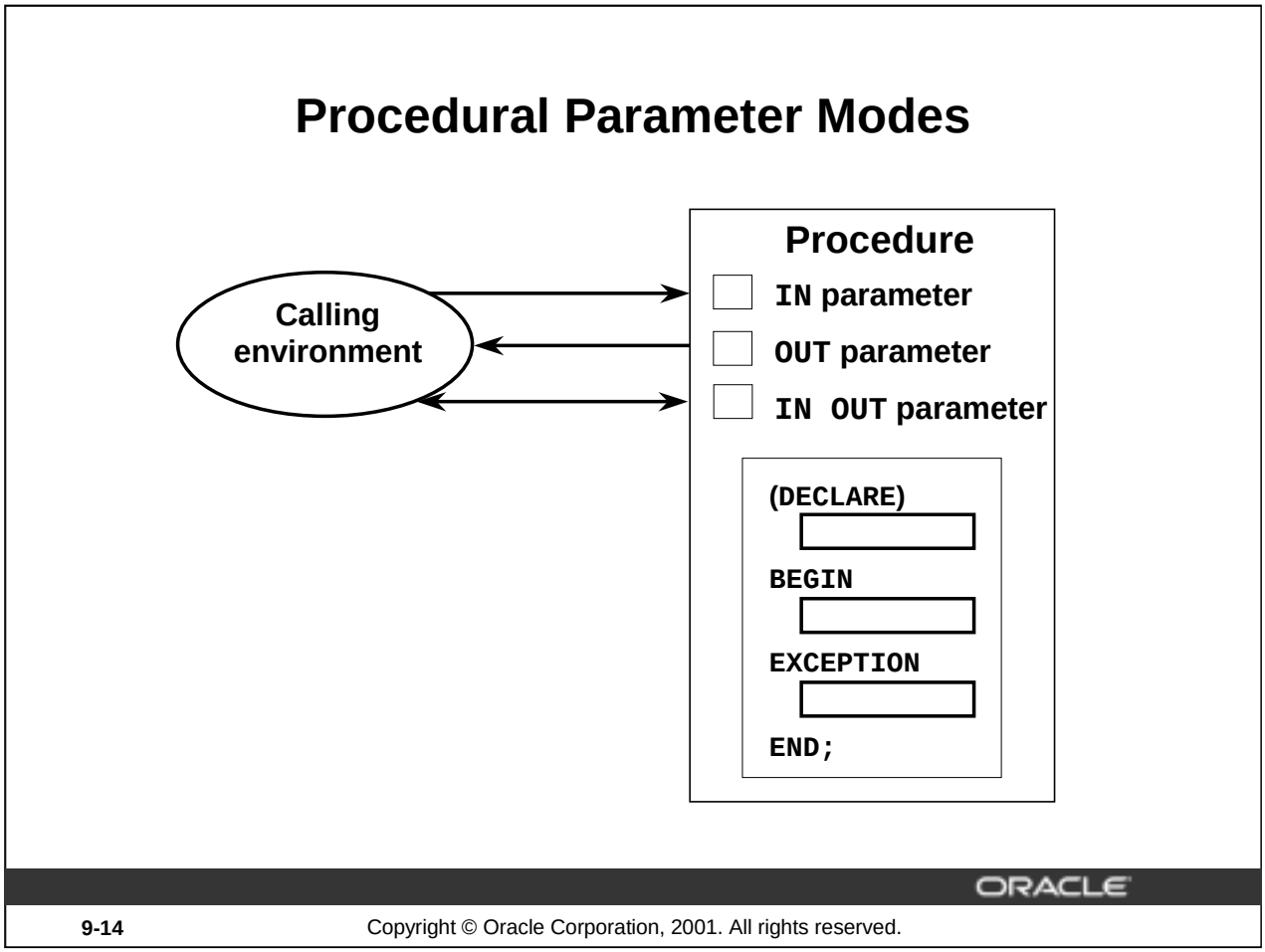
Formal parameters are variables declared in the parameter list of a subprogram specification. For example, in the procedure `RAISE_SAL`, the variables `P_ID` and `P_AMOUNT` are formal parameters.

Actual parameters are variables or expressions referenced in the parameter list of a subprogram call. For example, in the call `raise_sal(v_id , 2000)` to the procedure `RAISE_SAL`, the variable `V_ID` and `2000` are actual parameters.

- Actual parameters are evaluated and results are assigned to formal parameters during the subprogram call.
- Actual parameters can also be expressions such as in the following:

```
raise_sal(v_id, raise+100);
```

- It is good practice to use different names for formal and actual parameters. Formal parameters have the prefix `p_` in this course.
- The formal and actual parameters should be of compatible data types. If necessary, before assigning the value, PL/SQL converts the data type of the actual parameter value to that of the formal parameter.



Procedural Parameter Modes

You can transfer values to and from the calling environment through parameters. Select one of the three modes for each parameter: `IN`, `OUT`, or `IN OUT`.

Attempts to change the value of an IN parameter will result in an error.

Note: `DATATYPE` can be only the `%TYPE` definition, the `%ROWTYPE` definition, or an explicit data type with no size specification.

Type of Parameter	Description
`IN` (default)	Passes a constant value from the calling environment into the procedure
`OUT`	Passes a value from the procedure to the calling environment
`IN OUT`	Passes a value from the calling environment into the procedure and a possibly different value from the procedure back to the calling environment using the same parameter

Creating Procedures with Parameters

IN	OUT	IN OUT
Default mode	Must be specified	Must be specified
Value is passed into subprogram	Returned to calling environment	Passed into subprogram; returned to calling environment
Formal parameter acts as a constant	Uninitialized variable	Initialized variable
Actual parameter can be a literal, expression, constant, or initialized variable	Must be a variable	Must be a variable
Can be assigned a default value	Cannot be assigned a default value	Cannot be assigned a default value

ORACLE

Creating Procedures with Parameters

When you create the procedure, the formal parameter defines the value used in the executable section of the PL/SQL block, whereas the actual parameter is referenced when invoking the procedure.

The parameter mode `IN` is the default parameter mode. That is, no mode is specified with a parameter, the parameter is considered an `IN` parameter. The parameter modes `OUT` and `IN OUT` must be explicitly specified in front of such parameters.

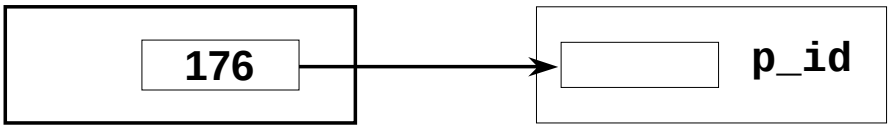
A formal parameter of `IN` mode cannot be assigned a value. That is, an `IN` parameter cannot be modified in the body of the procedure.

An `OUT` or `IN OUT` parameter must be assigned a value before returning to the calling environment.

`IN` parameters can be assigned a default value in the parameter list. `OUT` and `IN OUT` parameters cannot be assigned default values.

By default, the `IN` parameter is passed by reference and the `OUT` and `IN OUT` parameters are passed by value. To improve performance with `OUT` and `IN OUT` parameters, the compiler hint `NOCOPY` can be used to request to pass by reference. Using `NOCOPY` is discussed in detail in the *Advanced PL/SQL* course.

IN Parameters: Example




```
CREATE OR REPLACE PROCEDURE raise_salary
  (p_id IN employees.employee_id%TYPE)
IS
BEGIN
  UPDATE employees
  SET    salary = salary * 1.10
  WHERE  employee_id = p_id;
END raise_salary;
/
```

```
Procedure created.
```

ORACLE

IN Parameters: Example

The example in the slide shows a procedure with one IN parameter. Running this statement in *i*SQL*Plus creates the RAISE_SALARY procedure. When invoked, RAISE_SALARY accepts the parameter for the employee ID and updates the employee's record with a salary increase of 10 percent.

To invoke a procedure in *i*SQL*Plus, use the EXECUTE command.

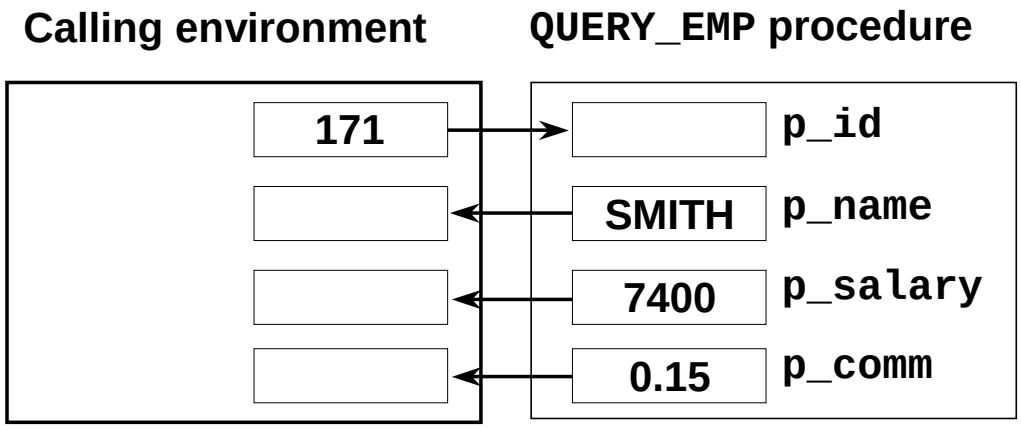
```
EXECUTE raise_salary (176)
```

To invoke a procedure from another procedure, use a direct call. At the location of calling the new procedure, enter the procedure name and actual parameters.

```
raise_salary (176);
```

IN parameters are passed as constants from the calling environment into the procedure. Attempts to change the value of an IN parameter result in an error.

OUT Parameters: Example



ORACLE

OUT Parameters: Example

In this example, you create a procedure with OUT parameters to retrieve information about an employee. The procedure accepts a value 171 for employee ID and retrieves the name, salary, and commission percentage of the employee with ID 171 into the three output parameters. The code to create the QUERY_EMP procedure is shown in the next slide.

OUT Parameters: Example

emp_query.sql

```
CREATE OR REPLACE PROCEDURE query_emp
  (p_id      IN   employees.employee_id%TYPE,
   p_name    OUT  employees.last_name%TYPE,
   p_salary  OUT  employees.salary%TYPE,
   p_comm    OUT  employees.commission_pct%TYPE)
IS
BEGIN
  SELECT   last_name, salary, commission_pct
   INTO     p_name, p_salary, p_comm
   FROM     employees
   WHERE    employee_id = p_id;
END query_emp;
/
```

ORACLE

OUT Parameters: Example (continued)

Run the script file shown in the slide to create the QUERY_EMP procedure. This procedure has four formal parameters. Three of them are OUT parameters that return values to the calling environment.

The procedure accepts an EMPLOYEE_ID value for the parameter P_ID. The name, salary, and commission percentage values corresponding to the employee ID are retrieved into the three OUT parameters whose values are returned to the calling environment.

Notice that the name of the script file need not be the same as the procedure name. (The script file is on the client side and the procedure is being stored on the database schema.)

Viewing OUT Parameters

- **Load and run the `emp_query.sql` script file to create the `QUERY_EMP` procedure.**
- **Declare host variables, execute the `QUERY_EMP` procedure, and print the value of the global variable `G_NAME`.**

```
VARIABLE g_name     VARCHAR2(25)
VARIABLE g_sal      NUMBER
VARIABLE g_comm     NUMBER

EXECUTE query_emp(171, :g_name, :g_sal, :g_comm)

PRINT g_name
```

```
PL/SQL procedure successfully completed.

G_NAME
Smith
```

ORACLE

How to View the Value of OUT Parameters with *i*SQL*Plus

1. Run the SQL script file to generate and compile the source code.
2. Create host variables in *i*SQL*Plus, using the `VARIABLE` command.
3. Invoke the `QUERY_EMP` procedure, supplying these host variables as the `OUT` parameters. Note the use of the colon (:) to reference the host variables in the `EXECUTE` command.
4. To view the values passed from the procedure to the calling environment, use the `PRINT` command.

The example in the slide shows the value of the `G_NAME` variable passed back to the the calling environment. The other variables can be viewed, either individually, as above, or with a single `PRINT` command.

```
PRINT g_name g_sal g_comm
```

Do not specify a size for a host variable of data type `NUMBER` when using the `VARIABLE` command. A host variable of data type `CHAR` or `VARCHAR2` defaults to a length of one, unless a value is supplied in parentheses.

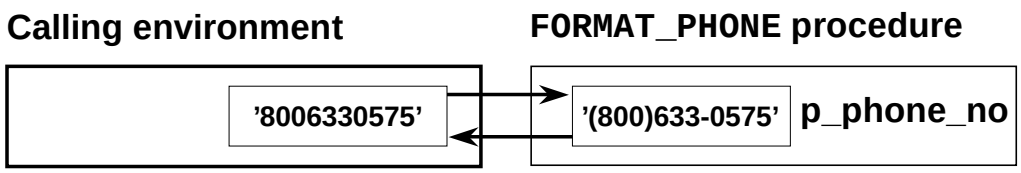
`PRINT` and `VARIABLE` are *i*SQL*Plus commands.

Note: Passing a constant or expression as an actual parameter to the `OUT` variable causes compilation errors. For Example:

```
EXECUTE query_emp(171, :g_name, raise+100, :g_comm)
```

causes a compilation error.

IN OUT Parameters




```
CREATE OR REPLACE PROCEDURE format_phone
  (p_phone_no IN OUT VARCHAR2)
IS
BEGIN
  p_phone_no := '(' || SUBSTR(p_phone_no,1,3) ||
                ')' || SUBSTR(p_phone_no,4,3) ||
                '-' || SUBSTR(p_phone_no,7);
END format_phone;
/
```

ORACLE

Using IN OUT Parameters

With an IN OUT parameter, you can pass values into a procedure and return a value to the calling environment. The value that is returned is either the original, an unchanged value, or a new value set within the procedure.

An IN OUT parameter acts as an initialized variable.

Example

Create a procedure with an IN OUT parameter to accept a character string containing 10 digits and return a phone number formatted as (800) 633-0575.

Run the statement to create the FORMAT_PHONE procedure.

Viewing IN OUT Parameters

```
VARIABLE g_phone_no VARCHAR2(15)
BEGIN
  :g_phone_no := '8006330575';
END;
/
PRINT g_phone_no
EXECUTE format_phone (:g_phone_no)
PRINT g_phone_no
```

PL/SQL procedure successfully completed.

G_PHONE_NO
8006330575

PL/SQL procedure successfully completed.

G_PHONE_NO
(800)633-0575

How to View IN OUT Parameters with *i*SQL*Plus

1. Create a host variable, using the VARIABLE command.
2. Populate the host variable with a value, using an anonymous PL/SQL block.
3. Invoke the FORMAT_PHONE procedure, supplying the host variable as the IN OUT parameter. Note the use of the colon (:) to reference the host variable in the EXECUTE command.
4. To view the value passed back to the calling environment, use the PRINT command.

Methods for Passing Parameters

- **Positional: List actual parameters in the same order as formal parameters.**
- **Named: List actual parameters in arbitrary order by associating each with its corresponding formal parameter.**
- **Combination: List some of the actual parameters as positional and some as named.**

ORACLE

Parameter Passing Methods

For a procedure that contains multiple parameters, you can use a number of methods to specify the values of the parameters.

Method	Description
Positional	Lists values in the order in which the parameters are declared
Named association	Lists values in arbitrary order by associating each one with its parameter name, using special syntax (=>)
Combination	Lists the first values positionally, and the remainder using the special syntax of the named method

DEFAULT Option for Parameters

```
CREATE OR REPLACE PROCEDURE add_dept
  (p_name  IN departments.department_name%TYPE
                          DEFAULT 'unknown',
   p_loc   IN departments.location_id%TYPE
                          DEFAULT 1700)
IS
BEGIN
  INSERT INTO departments(department_id,
           department_name, location_id)
  VALUES (departments_seq.NEXTVAL, p_name, p_loc);
END add_dept;
/
```

```
Procedure created.
```

ORACLE

Example of Default Values for Parameters

You can initialize IN parameters to default values. That way, you can pass different numbers of actual parameters to a subprogram, accepting or overriding the default values as you please. Moreover, you can add new formal parameters without having to change every call to the subprogram.

Execute the statement in the slide to create the `ADD_DEPT` procedure. Note the use of the `DEFAULT` clause in the declaration of the formal parameter.

You can assign default values only to parameters of the `IN` mode. `OUT` and `IN OUT` parameters are not permitted to have default values. If default values are passed to these types of parameters, you get the following compilation error:

```
PLS-00230: OUT and IN OUT formal parameters may not have default
expressions
```

If an actual parameter is not passed, the default value of its corresponding formal parameter is used. Consider the calls to the above procedure that are depicted in the next page.

Examples of Passing Parameters

```
BEGIN
  add_dept;
  add_dept ('TRAINING', 2500);
  add_dept ( p_loc => 2400, p_name =>'EDUCATION');
  add_dept ( p_loc => 1200) ;
END;
/
SELECT department_id, department_name, location_id
FROM departments;
```

ORACLE

Example of Default Values for Parameters (continued)

The anonymous block above shows the different ways the ADD_DEPT procedure can be invoked, and the output of each way the procedure is invoked.

Usually, you can use positional notation to override the default values of formal parameters. However, you cannot skip a formal parameter by leaving out its actual parameter.

Note: All the positional parameters should precede the named parameters in a subprogram call. Otherwise, you will receive an error message, as shown in the following example:

```
EXECUTE add_dept(p_name=>'new dept', 'new location')
BEGIN add_dept(p_name=>'new dept', 'new location'); END;
                                   *
ERROR at line 1:
ORA-06550: line 1, column 31:
PLS-00312: a positional parameter association may not follow a named
association
ORA-06550: line 1, column 7:
PL/SQL: Statement ignored
```

Declaring Subprograms

leave_emp2.sql

```
CREATE OR REPLACE PROCEDURE leave_emp2
  (p_id  IN  employees.employee_id%TYPE)
IS
  PROCEDURE log_exec
  IS
  BEGIN
    INSERT INTO log_table (user_id, log_date)
    VALUES (USER, SYSDATE);
  END log_exec;
BEGIN
  DELETE FROM employees
  WHERE employee_id = p_id;
  log_exec;
END leave_emp2;
/
```

ORACLE

Declaring Subprograms

You can declare subprograms in any PL/SQL block. This is an alternative to creating the stand-alone procedure `LOG_EXEC`. Subprograms declared in this manner are called local subprograms (or local modules). Because they are defined within a declaration section of another program, the scope of local subprograms is limited to the parent (enclosing) block in which they are defined. This means that local subprograms cannot be called from outside the block in which they are declared. Declaring local subprograms enhances the clarity of the code by assigning appropriate business-rule identifiers to blocks of code.

Note: You must declare the subprogram in the declaration section of the block, and it must be the last item, after all the other program items. For example, a variable declared after the end of the subprogram, before the `BEGIN` of the procedure, will cause a compilation error.

If the code must be accessed by multiple applications, place the subprogram in a package or create a stand-alone subprogram with the code. Packages are discussed later in this course.

Invoking a Procedure from an Anonymous PL/SQL Block

```
DECLARE
  v_id NUMBER := 163;
BEGIN
  raise_salary(v_id);    --invoke procedure
  COMMIT;
...
END;
```

ORACLE

Invoking a Procedure from an Anonymous PL/SQL Block

Invoke the RAISE_SALARY procedure from an anonymous PL/SQL block, as shown in the slide.

Procedures are callable from *any* tool or language that supports PL/SQL.

You have already seen how to invoke an independent procedure from *i*SQL*Plus.

Invoking a Procedure from Another Procedure

process_emps.sql

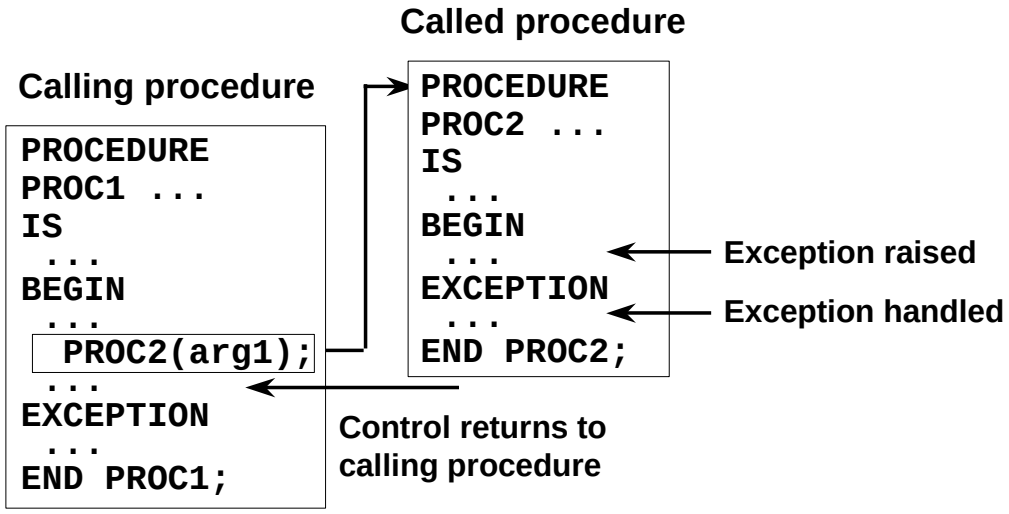
```
CREATE OR REPLACE PROCEDURE process_emps
IS
   CURSOR emp_cursor IS
    SELECT employee_id
    FROM   employees;
BEGIN
   FOR emp_rec IN emp_cursor
   LOOP
    raise_salary(emp_rec.employee_id);
   END LOOP;
   COMMIT;
END process_emps;
/
```

ORACLE

Invoking a Procedure from Another Procedure

This example shows you how to invoke a procedure from another stored procedure. The PROCESS_EMPS stored procedure uses a cursor to process all the records in the EMPLOYEES table and passes each employee's ID number to the RAISE_SALARY procedure, which results in a 10 percent salary increase across the company.

Handled Exceptions



ORACLE

How Handled Exceptions Affect the Calling Procedure

When you develop procedures that are called from other procedures, you should be aware of the effects that handled and unhandled exceptions have on the transaction and the calling procedure.

When an exception is raised in a called procedure, control immediately goes to the exception section of that block. If the exception is handled, the block terminates, and control goes to the calling program. Any data manipulation language (DML) statements issued before the exception was raised remain as part of the transaction.

Handled Exceptions

```
CREATE PROCEDURE p2_ins_dept(p_locid NUMBER) IS
 v_did NUMBER(4);
BEGIN
 DBMS_OUTPUT.PUT_LINE('Procedure p2_ins_dept started');
 INSERT INTO departments VALUES (5, 'Dept 5', 145, p_locid);
 SELECT department_id INTO v_did FROM employees
  WHERE employee_id = 999;
END;
```

```
CREATE PROCEDURE p1_ins_loc(p_lid NUMBER, p_city VARCHAR2)
IS
 v_city VARCHAR2(30); v_dname VARCHAR2(30);
BEGIN
 DBMS_OUTPUT.PUT_LINE('Main Procedure p1_ins_loc');
 INSERT INTO locations (location_id, city) VALUES (p_lid, p_city);
 SELECT city INTO v_city FROM locations WHERE location_id = p_lid;
 DBMS_OUTPUT.PUT_LINE('Inserted city '||v_city);
 DBMS_OUTPUT.PUT_LINE('Invoking the procedure p2_ins_dept ...');
 p2_ins_dept(p_lid);
EXCEPTION
  WHEN NO_DATA_FOUND THEN
   DBMS_OUTPUT.PUT_LINE('No such dept/loc for any employee');
END;
```

ORACLE

How Handled Exceptions Affect the Calling Procedure (continued)

The example in the slide shows two procedures. Procedure `P1_INS_LOC` inserts a new location (supplied through the parameters) into the `LOCATIONS` table. Procedure `P2_INS_DEPT` inserts a new department (with department ID 5) at the new location inserted through the `P1_INS_LOC` procedure. The `P1_INS_LOC` procedure invokes the `P2_INS_DEPT` procedure.

The `P2_INS_DEPT` procedure has a `SELECT` statement that selects `DEPARTMENT_ID` for a nonexisting employee and raises a `NO_DATA_FOUND` exception. Because this exception is not handled in the `P2_INS_DEPT` procedure, the control returns to the calling procedure `P1_INS_LOC` where the exception is handled. As the exception is handled, the DML in the `P2_INS_DEPT` procedure is not rolled back and is part of the transaction of the `P1_INS_LOC` procedure.

The following code shows that the `INSERT` statements from both the procedures are successful:

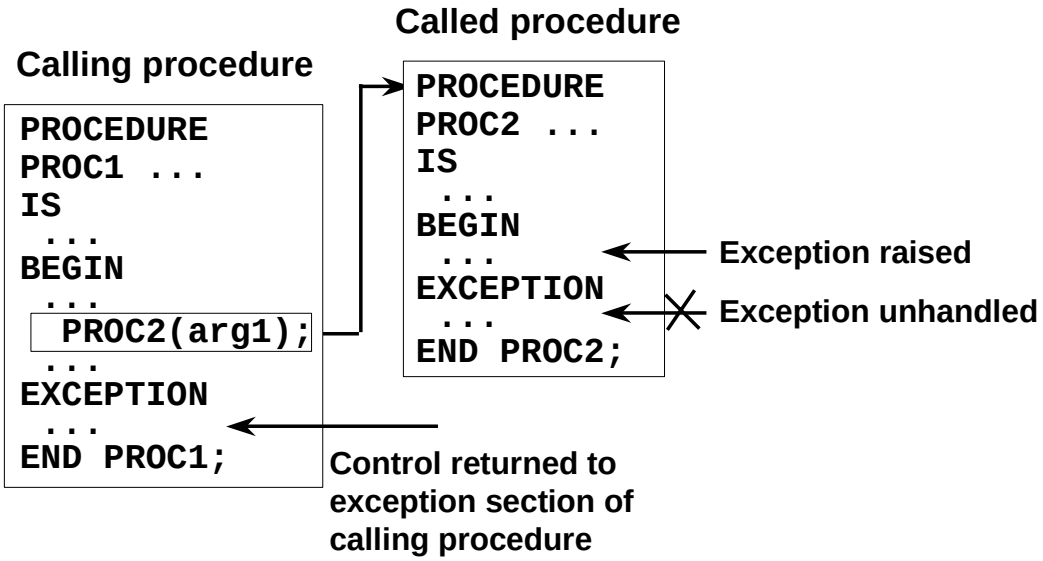
```
EXECUTE p1_ins_loc(1, 'Redwood Shores')
SELECT location_id, city FROM locations
 WHERE location_id = 1;
SELECT * FROM departments WHERE department_id = 5;
```

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

LOCATION_ID	CITY
1	Redwood Shores

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
5	Dept 5	145	1

Unhandled Exceptions



ORACLE

How Unhandled Exceptions Affect the Calling Procedure (continued)

When an exception is raised in a called procedure, control immediately goes to the exception section of that block. If the exception is unhandled, the block terminates, and control goes to the exception section of the calling procedure. PL/SQL does not roll back database work that is done by the subprogram.

If the exception is handled in the calling procedure, all DML statements in the calling procedure and in the called procedure remain as part of the transaction.

If the exception is unhandled in the calling procedure, the calling procedure terminates and the exception propagates to the calling environment. All the DML statements in the calling procedure and the called procedure are rolled back along with any changes to any host variables. The host environment determines the outcome for the unhandled exception.

Unhandled Exceptions

```
CREATE PROCEDURE p2_noexcep(p_locid NUMBER) IS
 v_did NUMBER(4);
BEGIN
 DBMS_OUTPUT.PUT_LINE('Procedure p2_noexcep started');
 INSERT INTO departments VALUES (6, 'Dept 6', 145, p_locid);
 SELECT department_id INTO v_did FROM employees
  WHERE employee_id = 999;
END;
```

```
CREATE PROCEDURE p1_noexcep(p_lid NUMBER, p_city VARCHAR2)
IS
 v_city VARCHAR2(30); v_dname VARCHAR2(30);
BEGIN
 DBMS_OUTPUT.PUT_LINE(' Main Procedure p1_noexcep');
 INSERT INTO locations (location_id, city) VALUES (p_lid, p_city);
 SELECT city INTO v_city FROM locations WHERE location_id = p_lid;
 DBMS_OUTPUT.PUT_LINE('Inserted new city '||v_city);
 DBMS_OUTPUT.PUT_LINE('Invoking the procedure p2_noexcep ...');
 p2_noexcep(p_lid);
END;
```

ORACLE

How Unhandled Exceptions Affect the Calling Procedure (continued)

The example in the slide shows two procedures. Procedure P1_NOEXCEP inserts a new location (supplied through the parameters) into the LOCATIONS table. Procedure P2_NOEXCEP inserts a new department (with department ID 5) at the new location inserted through the P1_NOEXCEP procedure. Procedure P1_NOEXCEP invokes the P2_NOEXCEP procedure.

The P2_NOEXCEP procedure has a SELECT statement that selects DEPARTMENT_ID for a nonexisting employee and raises a NO_DATA_FOUND exception. Because this exception is not handled in the P2_NOEXCEP procedure, the control returns to the calling procedure P1_NOEXCEP. The exception is not handled. Because the exception is not handled, the DML in the P2_NOEXCEP procedure is rolled back along with the transaction of the P1_NOEXCEP procedure.

The following code shows that the DML statements from both the procedures are unsuccessful.

```
EXECUTE p1_noexcep(3, 'New Delhi')
SELECT location_id, city FROM locations
 WHERE location_id = 3;
SELECT * FROM departments WHERE department_id = 6;
```

Removing Procedures

Drop a procedure stored in the database.

Syntax:

```
DROP PROCEDURE procedure_name
```

Example:

```
DROP PROCEDURE raise_salary;
```

```
Procedure dropped
```

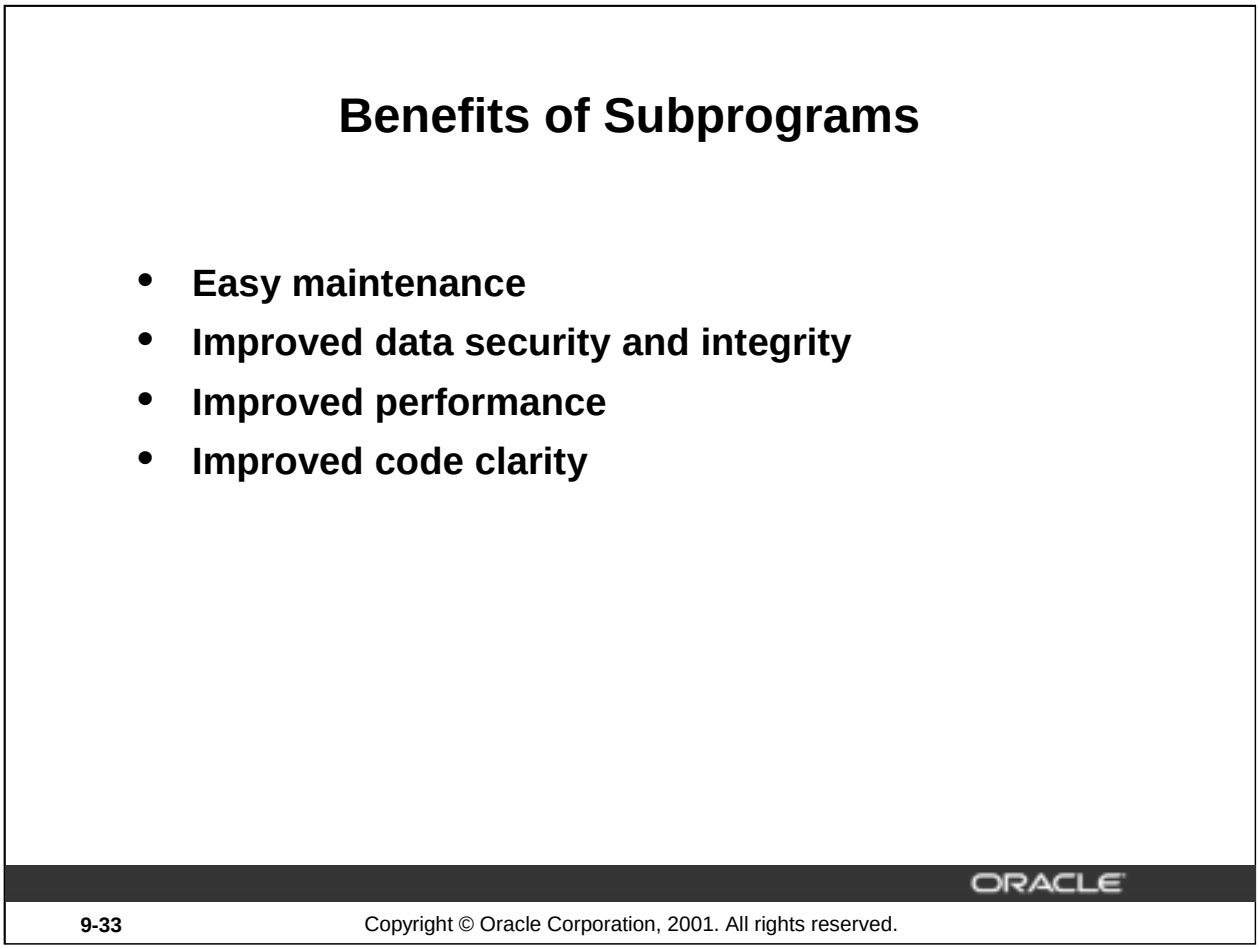
ORACLE

Removing Procedures

When a stored procedure is no longer required, you can use a SQL statement to drop it.

To remove a server-side procedure by using *i*SQL*Plus, execute the SQL command `DROP PROCEDURE`.

Issuing rollback does not have an effect after executing a data definition language (DDL) command such as `DROP PROCEDURE`, which commits any pending transactions.



Benefits of Subprograms

Procedures and functions have many benefits in addition to modularizing application development:

- Easy maintenance: Subprograms are located in one location and hence it is easy to:
  - Modify routines online without interfering with other users
  - Modify one routine to affect multiple applications
  - Modify one routine to eliminate duplicate testing
- Improved data security and integrity
  - Controls indirect access to database objects from nonprivileged users with security privileges. As a subprogram is executed with its definer's right by default, it is easy to restrict the access privilege by granting a privilege only to execute the subprogram to a user.
  - Ensures that related actions are performed together, or not at all, by funneling activity for related tables through a single path
- Improved performance
  - After a subprogram is compiled, the parsed code is available in the shared SQL area of the server and subsequent calls to the subprogram use this parsed code. This avoids reparsing for multiple users.
  - Avoids PL/SQL parsing at run time by parsing at compile time
  - Reduces the number of calls to the database and decreases network traffic by bundling commands
- Improves code clarity: Using appropriate identifier names to describe the action of the routines reduces the need for comments and enhances the clarity of the code.

Summary

In this lesson, you should have learned that:

- **A subprogram is a named PL/SQL block that can accept parameters and be invoked.**
- **A procedure is a subprogram that performs an action.**
- **You create procedures by using the `CREATE PROCEDURE` command.**
- **You can compile and save a procedure in the database.**
- **Parameters are used to pass data from the calling environment to the procedure.**
- **There are three parameter modes: `IN`, `OUT`, and `IN OUT`.**

ORACLE

Summary

A subprogram is a named PL/SQL block that can accept parameters and be invoked from a calling environment.

A procedure is a subprogram that performs a specified action. You can compile and save a procedure as stored procedure in the database. A procedure can return zero or more values through its parameters to its calling environment. There are three parameter modes `IN`, `OUT`, and `IN OUT`.

Summary

- Local subprograms are programs that are defined within the declaration section of another program.
- Procedures can be invoked from any tool or language that supports PL/SQL.
- You should be aware of the effect of handled and unhandled exceptions on transactions and calling procedures.
- You can remove procedures from the database by using the `DROP PROCEDURE` command.
- Procedures can serve as building blocks for an application.

ORACLE

Summary (continued)

Subprograms that are defined within the declaration section of another program are called local subprograms. The scope of the local subprograms is the program unit within which it is defined.

You should be aware of the effect of handled and unhandled exceptions on transactions and calling procedures. The exceptions are handled in the exception section of a subprogram.

You can modify and remove procedures. You can also create client-side procedures that can be used by client-side applications.

Practice 9 Overview

This practice covers the following topics:

- **Creating stored procedures to:**
  - **Insert new rows into a table, using the supplied parameter values**
  - **Update data in a table for rows matching with the supplied parameter values**
  - **Delete rows from a table that match the supplied parameter values**
  - **Query a table and retrieve data based on supplied parameter values**
- **Handling exceptions in procedures**
- **Compiling and invoking procedures**

ORACLE

Practice 9 Overview

In this practice you create procedures that issue DML and query commands.

If you encounter compilation errors when you are using *i*SQL*Plus, use the `SHOW ERRORS` command. Using the `SHOW ERRORS` command is discussed in detail in the *Managing Subprograms* lesson.

If you correct any compilation errors in *i*SQL*Plus, do so in the original script file, not in the buffer, and then rerun the new version of the file. This will save a new version of the procedure to the data dictionary.

Practice 9

Note: You can find table descriptions and sample data in Appendix D "Table Descriptions and Data."

Save your subprograms as `.sql` files, using the `Save Script` button.

Remember to set the `SERVEROUTPUT` on if you set it off previously.

1. Create and invoke the `ADD_JOB` procedure and consider the results.
   a. Create a procedure called `ADD_JOB` to insert a new job into the `JOBS` table. Provide the ID and title of the job, using two parameters.
   b. Compile the code, and invoke the procedure with `IT_DBA` as job ID and `Database Administrator` as job title. Query the `JOBS` table to view the results.

JOB_ID	JOB_TITLE	MIN_SALARY	MAX_SALARY
IT_DBA	Database Administrator		

   c. Invoke your procedure again, passing a job ID of `ST_MAN` and a job title of `Stock Manager`. What happens and why?

_____

_____

2. Create a procedure called `UPD_JOB` to modify a job in the `JOBS` table.
   a. Create a procedure called `UPD_JOB` to update the job title. Provide the job ID and a new title, using two parameters. Include the necessary exception handling if no update occurs.
   b. Compile the code; invoke the procedure to change the job title of the job ID `IT_DBA` to `Data Administrator`. Query the `JOBS` table to view the results.

JOB_ID	JOB_TITLE	MIN_SALARY	MAX_SALARY
IT_DBA	Data Administrator		

   Also check the exception handling by trying to update a job that does not exist (you can use job ID `IT_WEB` and job title `Web Master`).

3. Create a procedure called `DEL_JOB` to delete a job from the `JOBS` table.
   a. Create a procedure called `DEL_JOB` to delete a job. Include the necessary exception handling if no job is deleted.
   b. Compile the code; invoke the procedure using job ID `IT_DBA`. Query the `JOBS` table to view the results.

```
no rows selected
```

   Also, check the exception handling by trying to delete a job that does not exist (use job ID `IT_WEB`). You should get the message you used in the exception-handling section of the procedure as output.

Practice 9 (continued)

4. Create a procedure called `QUERY_EMP` to query the `EMPLOYEES` table, retrieving the salary and job ID for an employee when provided with the employee ID.
   a. Create a procedure that returns a value from the `SALARY` and `JOB_ID` columns for a specified employee ID.

      Use host variables for the two `OUT` parameters salary and job ID.

   b. Compile the code, invoke the procedure to display the salary and job ID for employee ID 120.

G_SAL
8000

G_JOB
ST_MAN

   c. Invoke the procedure again, passing an `EMPLOYEE_ID` of 300. What happens and why?

      _____

      _____

10

Creating Functions

ORACLE

Objectives

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

- **Describe the uses of functions**
- **Create stored functions**
- **Invoke a function**
- **Remove a function**
- **Differentiate between a procedure and a function**

ORACLE

Lesson Aim

In this lesson, you will learn how to create and invoke functions.

Overview of Stored Functions

- **A function is a named PL/SQL block that returns a value.**
- **A function can be stored in the database as a schema object for repeated execution.**
- **A function is called as part of an expression.**

ORACLE

Stored Functions

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

Functions can be stored in the database as a schema object for repeated execution. A function stored in the database is referred to as a stored function. Functions can also be created at client side applications. This lesson discusses creating stored functions. Refer to appendix “Using Procedure Builder” for creating client-side applications.

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

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

Syntax for Creating Functions

```
CREATE [OR REPLACE] FUNCTION function_name
 [(parameter1 [mode1] datatype1,
  parameter2 [mode2] datatype2,
  . . .)]
RETURN datatype
IS|AS
PL/SQL Block;
```

The PL/SQL block must have at least one RETURN statement.

ORACLE

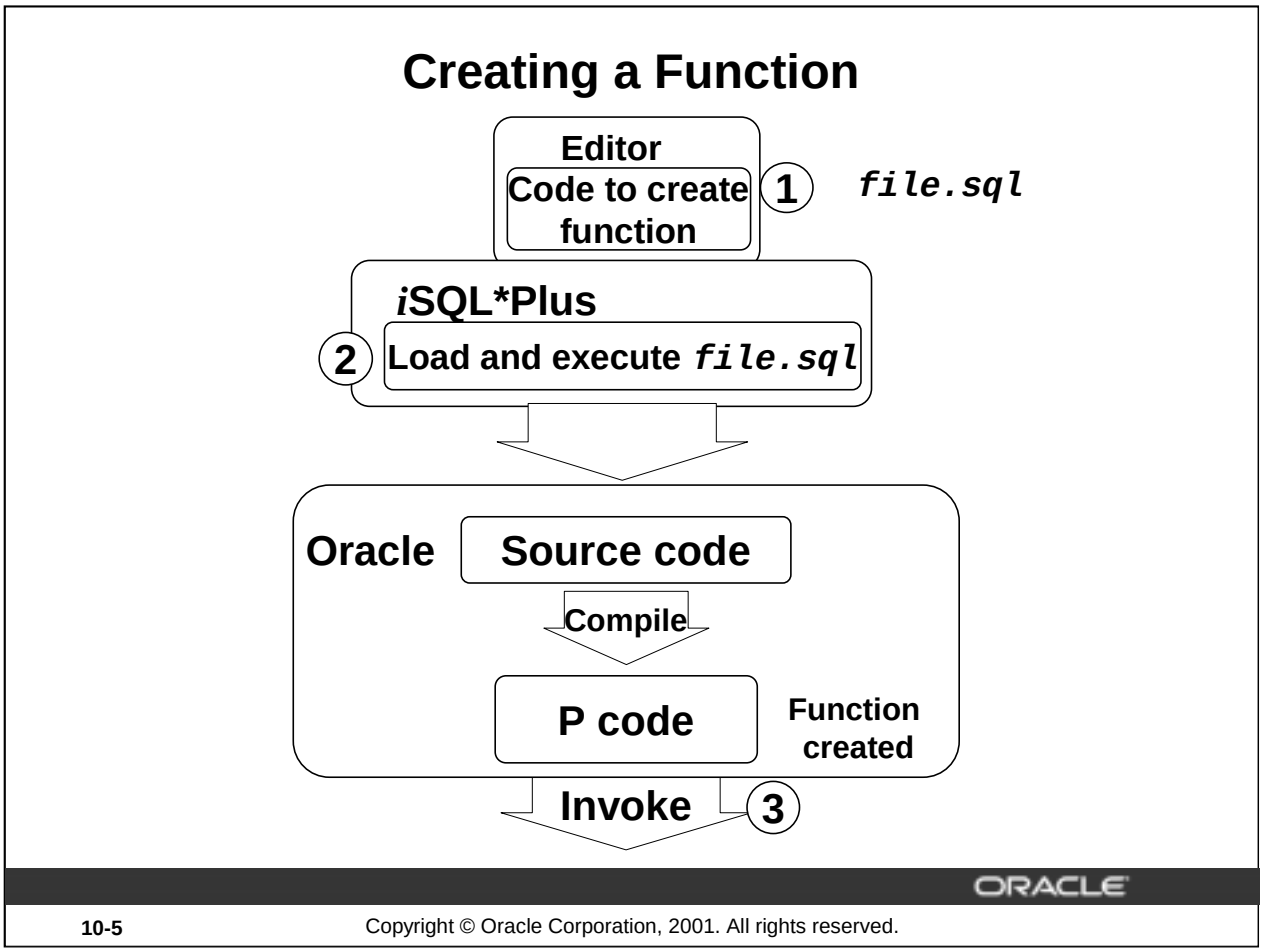
Creating Functions Syntax

A function is a PL/SQL block that returns a value. You create new functions with the CREATE FUNCTION statement, which may declare a list of parameters, must return one value, and must define the actions to be performed by the standard PL/SQL block.

- The REPLACE option indicates that if the function exists, it will be dropped and replaced with the new version created by the statement.
- The RETURN data type must not include a size specification.
- PL/SQL blocks start with either BEGIN or the declaration of local variables and end with either END or END *function_name*. There must be at least one RETURN (*expression*) statement. You cannot reference host or bind variables in the PL/SQL block of a stored function.

Syntax Definitions

Parameter	Description
function_name	Name of the function
parameter	Name of a PL/SQL variable whose value is passed into the function
mode	The type of the parameter; only IN parameters should be declared
datatype	Data type of the parameter
RETURN *datatype*	Data type of the RETURN value that must be output by the function
PL/SQL block	Procedural body that defines the action performed by the function



How to Develop Stored Functions

The following are the basic steps you use to develop a stored function. The next two pages provide further details about creating functions.

1. Write the syntax: Enter the code to create a function in a text editor and save it as a SQL script file.
2. Compile the code: Using *i*SQL*Plus, upload and run the SQL script file. The source code is compiled into P code. The function is created.
3. Invoke the function from a PL/SQL block.

Returning a Value

- Add a `RETURN` clause with the data type in the header of the function.
- Include one `RETURN` statement in the executable section.

Although multiple `RETURN` statements are allowed in a function (usually within an `IF` statement), only one `RETURN` statement is executed, because after the value is returned, processing of the block ceases.

Note: The PL/SQL compiler generates the *pseudocode* or P code, based on the parsed code. The PL/SQL engine executes this when the procedure is invoked.

Creating a Stored Function by Using *i*SQL*Plus

1. Enter the text of the `CREATE FUNCTION` statement in an editor and save it as a SQL script file.
2. Run the script file to store the source code and compile the function.
3. Use `SHOW ERRORS` to see compilation errors.
4. When successfully compiled, invoke the function.

ORACLE

How to Create a Stored Function

1. Enter the text of the `CREATE FUNCTION` statement in a system editor or word processor and save it as a script file (`.sql` extension).
2. From *i*SQL*Plus, load and run the script file to store the source code and compile the source code into P-code.
3. Use `SHOW ERRORS` to see any compilation errors.
4. When the code is successfully compiled, the function is ready for execution. Invoke the function from an Oracle server environment.

A script file with the `CREATE FUNCTION` statement enables you to change the statement if compilation or run-time errors occur, or to make subsequent changes to the statement. You cannot successfully invoke a function that contains any compilation or run-time errors. In *i*SQL*Plus, use `SHOW ERRORS` to see any compilation errors.

Running the `CREATE FUNCTION` statement stores the source code in the data dictionary even if the function contains compilation errors.

Note: If there are any compilation errors and you make subsequent changes to the `CREATE FUNCTION` statement, you either have to drop the function first or use the `OR REPLACE` syntax.

Creating a Stored Function by Using *i*SQL*Plus: Example

get_salary.sql

```
CREATE OR REPLACE FUNCTION get_sal
      (p_id  IN employees.employee_id%TYPE)
      RETURN NUMBER
    IS
      v_salary employees.salary%TYPE :=0;
    BEGIN
      SELECT salary
      INTO   v_salary
      FROM   employees
      WHERE  employee_id = p_id;
     RETURN v_salary;
END get_sal;
/
```

ORACLE

Example

Create a function with one `IN` parameter to return a number.

Run the script file to create the `GET_SAL` function. Invoke a function as part of a PL/SQL expression, because the function will return a value to the calling environment.

It is a good programming practice to assign a returning value to a variable and use a single `RETURN` statement in the executable section of the code. There can be a `RETURN` statement in the exception section of the program also.

Executing Functions

- **Invoke a function as part of a PL/SQL expression.**
- **Create a variable to hold the returned value.**
- **Execute the function. The variable will be populated by the value returned through a `RETURN` statement.**

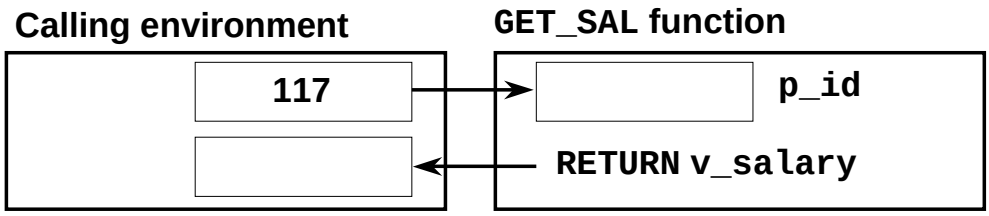
ORACLE

Function Execution

A function may accept one or many parameters, but must return a single value. You invoke functions as part of PL/SQL expressions, using variables to hold the returned value.

Although the three parameter modes, `IN` (the default), `OUT`, and `IN OUT`, can be used with any subprogram, avoid using the `OUT` and `IN OUT` modes with functions. The purpose of a function is to accept zero or more arguments (actual parameters) and return a single value. To have a function return multiple values is poor programming practice. Also, functions should be free from side effects, which change the values of variables that are not local to the subprogram. Side effects are discussed later in this lesson.

Executing Functions: Example



1. Load and run the `get_salary.sql` file to create the function

```
2 → VARIABLE g_salary NUMBER
3 → EXECUTE :g_salary := get_sal(117)
4 → PRINT g_salary
```

```
PL/SQL procedure successfully completed.

G_SALARY
2800
```

ORACLE

Example

Execute the GET_SAL function from *i*SQL*Plus:

1. Load and run the script file `get_salary.sql` to create the stored function GET_SAL.
2. Create a host variable that will be populated by the RETURN `(variable)` statement within the function.
3. Using the EXECUTE command in *i*SQL*Plus, invoke the GET_SAL function by creating a PL/SQL expression. Supply a value for the parameter (employee ID in this example). The value returned from the function will be held by the host variable, G_SALARY. Note the use of the colon (:) to reference the host variable.
4. View the result of the function call by using the PRINT command. Employee Tobias, with employee ID 117, earns a monthly salary of 2800.

In a function, there must be at least one execution path that leads to a RETURN statement. Otherwise, you get a `Function returned without value` error at run time.

Advantages of User-Defined Functions in SQL Expressions

- **Extend SQL where activities are too complex, too awkward, or unavailable with SQL**
- **Can increase efficiency when used in the `WHERE` clause to filter data, as opposed to filtering the data in the application**
- **Can manipulate character strings**

ORACLE

Invoking User-Defined Functions from SQL Expressions

SQL expressions can reference PL/SQL user-defined functions. Anywhere a built-in SQL function can be placed, a user-defined function can be placed as well.

Advantages

- Permits calculations that are too complex, awkward, or unavailable with SQL
- Increases data independence by processing complex data analysis within the Oracle server, rather than by retrieving the data into an application
- Increases efficiency of queries by performing functions in the query rather than in the application
- Manipulates new types of data (for example, latitude and longitude) by encoding character strings and using functions to operate on the strings

Invoking Functions in SQL Expressions: Example

```
CREATE OR REPLACE FUNCTION tax(p_value IN NUMBER)
 RETURN NUMBER IS
BEGIN
   RETURN (p_value * 0.08);
END tax;
/
SELECT employee_id, last_name, salary, tax(salary)
FROM   employees
WHERE  department_id = 100;
```

Function created.

EMPLOYEE_ID	LAST_NAME	SALARY	TAX(SALARY)
108	Greenberg	12000	960
109	Faviet	9000	720
110	Chen	8200	656
111	Sciarra	7700	616
112	Urman	7800	624
113	Popp	6900	552

6 rows selected.

ORACLE

Example

The slide shows how to create a function `tax` that is invoked from a `SELECT` statement. The function accepts a `NUMBER` parameter and returns the tax after multiplying the parameter value with 0.08.

In *i*SQL*Plus, invoke the `TAX` function inside a query displaying employee ID, name, salary, and tax.

Locations to Call User-Defined Functions

- Select list of a `SELECT` command
- Condition of the `WHERE` and `HAVING` clauses
- `CONNECT BY`, `START WITH`, `ORDER BY`, and `GROUP BY` clauses
- `VALUES` clause of the `INSERT` command
- `SET` clause of the `UPDATE` command

ORACLE

Usage of User-Defined Functions

PL/SQL user-defined functions can be called from any SQL expression where a built-in function can be called.

Example:

```
SELECT  employee_id, tax(salary)
FROM    employees
WHERE tax(salary)>(SELECT MAX(tax(salary))
                   FROM employees WHERE department_id = 30)
ORDER BY tax(salary) DESC;
```

EMPLOYEE_ID	TAX(SALARY)
100	1920
101	1360
102	1360
145	1120
146	1080
201	1040
108	960
147	960
205	960
168	920

10 rows selected.

Restrictions on Calling Functions from SQL Expressions

To be callable from SQL expressions, a user-defined function must:

- **Be a stored function**
- **Accept only `IN` parameters**
- **Accept only valid SQL data types, not PL/SQL specific types, as parameters**
- **Return data types that are valid SQL data types, not PL/SQL specific types**

ORACLE

Restrictions When Calling Functions from SQL Expressions

To be callable from SQL expressions, a user-defined PL/SQL function must meet certain requirements.

- Parameters to a PL/SQL function called from a SQL statement must use positional notation. Named notation is not supported.
- Stored PL/SQL functions cannot be called from the `CHECK` constraint clause of a `CREATE` or `ALTER TABLE` command or be used to specify a default value for a column.
- You must own or have the `EXECUTE` privilege on the function to call it from a SQL statement.
- The functions must return data types that are valid SQL data types. They cannot be PL/SQL-specific data types such as `BOOLEAN`, `RECORD`, or `TABLE`. The same restriction applies to parameters of the function.

Note: Only stored functions are callable from SQL statements. Stored procedures cannot be called.

The ability to use a user-defined PL/SQL function in a SQL expression is available with PL/SQL 2.1 and later. Tools using earlier versions of PL/SQL do not support this functionality. Prior to Oracle9*i*, user-defined functions can be only single-row functions. Starting with Oracle9*i*, user-defined functions can also be defined as aggregate functions.

Note: Functions that are callable from SQL expressions cannot contain `OUT` and `IN OUT` parameters. Other functions can contain parameters with these modes, but it is not recommended.

Restrictions on Calling Functions from SQL Expressions

- **Functions called from SQL expressions cannot contain DML statements.**
- **Functions called from `UPDATE/DELETE` statements on a table T cannot contain DML on the same table T.**
- **Functions called from a DML statement on a table T cannot query the same table.**
- **Functions called from SQL statements cannot contain statements that end the transactions.**
- **Calls to subprograms that break the previous restriction are not allowed in the function.**

ORACLE

Controlling Side Effects

To execute a SQL statement that calls a stored function, the Oracle server must know whether the function is free of side effects. Side effects are unacceptable changes to database tables. Therefore, restrictions apply to stored functions that are called from SQL expressions.

Restrictions

- When called from a `SELECT` statement or a parallelized `UPDATE` or `DELETE` statement, the function cannot modify any database tables
- When called from an `UPDATE`, or `DELETE` statement, the function cannot query or modify any database tables modified by that statement.
- When called from a `SELECT`, `INSERT`, `UPDATE`, or `DELETE` statement, the function cannot execute SQL transaction control statements (such as `COMMIT`), session control statements (such as `SET ROLE`), or system control statements (such as `ALTER SYSTEM`). Also, it cannot execute DDL statements (such as `CREATE`) because they are followed by an automatic commit.
- The function cannot call another subprogram that breaks one of the above restrictions.

Restrictions on Calling from SQL

```
CREATE OR REPLACE FUNCTION dml_call_sql (p_sal NUMBER)
   RETURN NUMBER IS
BEGIN
   INSERT INTO employees(employee_id, last_name, email,
                         hire_date, job_id, salary)
        VALUES(1, 'employee 1', 'emp1@company.com',
                  SYSDATE, 'SA_MAN', 1000);
   RETURN (p_sal + 100);
END;
/
```

```
Function created.
```

```
UPDATE employees SET salary = dml_call_sql(2000)
 WHERE employee_id = 170;
```

```
UPDATE employees SET salary = dml_call_sql(2000)
                   *
ERROR at line 1:
ORA-04091: table HR.EMPLOYEES is mutating, trigger/function may not see it
ORA-06512: at "HR.DML_CALL_SQL", line 4
ORA-06512: at line 1
```

ORACLE

Restrictions on Calling Functions from SQL: Example

The code example in the slide shows an example of having a DML statement in a function. The function DML_CALL_SQL contains a DML statement that inserts a new record into the EMPLOYEES table. This function is invoked in the UPDATE statement that modifies the salary of employee 170 to the amount returned from the function. The UPDATE statement returns an error saying that the table is mutating.

Consider the following example where the function QUERY_CALL_SQL queries the SALARY column of the EMPLOYEE table:

```
CREATE OR REPLACE FUNCTION query_call_sql(a NUMBER)
  RETURN NUMBER IS
  s NUMBER;
BEGIN
  SELECT salary INTO s FROM employees
   WHERE employee_id = 170;
  RETURN (s + a);
END;
/
```

The above function, when invoked from the following UPDATE statement, returns the error message as shown in the slide.

```
UPDATE employees SET salary = query_call_sql(100)
 WHERE employee_id = 170;
```

Removing Functions

Drop a stored function.

Syntax:

```
DROP FUNCTION function_name
```

Example:

```
DROP FUNCTION get_sal;
```

```
Function dropped.
```

- **All the privileges granted on a function are revoked when the function is dropped.**
- **The `CREATE OR REPLACE` syntax is equivalent to dropping a function and recreating it. Privileges granted on the function remain the same when this syntax is used.**

ORACLE

Removing Functions

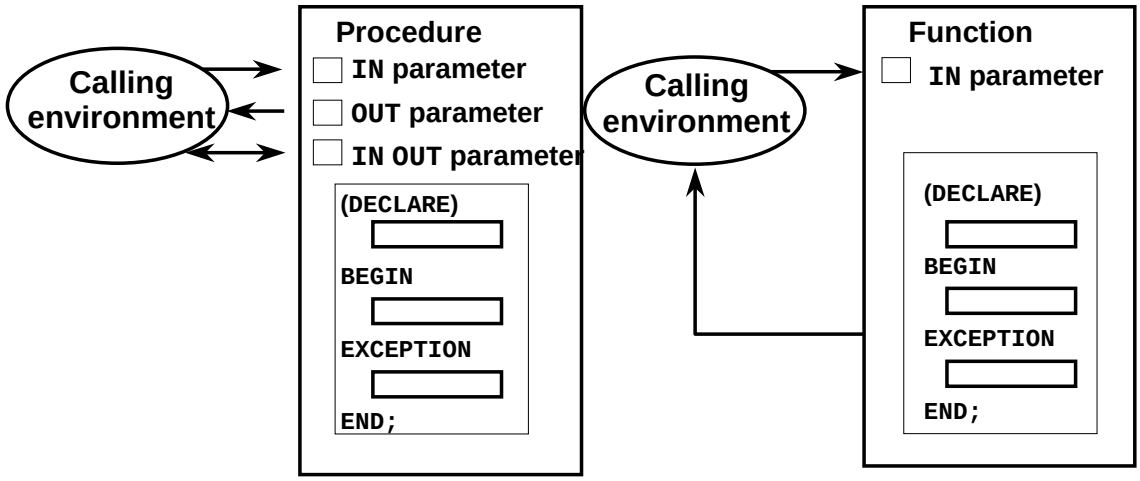
When a stored function is no longer required, you can use a SQL statement in *i*SQL*Plus to drop it.

To remove a stored function by using *i*SQL*Plus, execute the SQL command `DROP FUNCTION`.

`CREATE OR REPLACE` Versus `DROP` and `CREATE`:

The `REPLACE` clause in the `CREATE OR REPLACE` syntax is equivalent to dropping a function and re-creating it. When you use the `CREATE OR REPLACE` syntax, the privileges granted on this object to other users remain the same. When you `DROP` a function and then create it again, all the privileges granted on this function are automatically revoked.

Procedure or Function?



ORACLE

How Procedures and Functions Differ

You create a procedure to store a series of actions for later execution. A procedure can contain zero or more parameters that can be transferred to and from the calling environment, but a procedure does not have to return a value.

You create a function when you want to compute a value, which must be returned to the calling environment. A function can contain zero or more parameters that are transferred from the calling environment. Functions should return only a single value, and the value is returned through a `RETURN` statement. Functions used in SQL statements cannot have `OUT` or `IN OUT` mode parameters.

Comparing Procedures and Functions

Procedure	Function
Execute as a PL/SQL statement	Invoke as part of an expression
No RETURN clause in the header	Must contain a RETURN clause in the header
Can return none, one, or many values	Must return a single value
Can contain a RETURN statement	Must contain at least one RETURN statement

ORACLE

How Procedures and Functions Differ (continued)

A procedure containing one OUT parameter can be rewritten as a function containing a RETURN statement.

Benefits of Stored Procedures and Functions

- **Improved performance**
- **Easy maintenance**
- **Improved data security and integrity**
- **Improved code clarity**

ORACLE

Benefits

In addition to modularizing application development, stored procedures and functions have the following benefits:

- Improved performance
  - Avoid reparsing for multiple users by exploiting the shared SQL area
  - Avoid PL/SQL parsing at run time by parsing at compile time
  - Reduce the number of calls to the database and decrease network traffic by bundling commands
- Easy maintenance
  - Modify routines online without interfering with other users
  - Modify one routine to affect multiple applications
  - Modify one routine to eliminate duplicate testing
- Improved data security and integrity
  - Control indirect access to database objects from nonprivileged users with security privileges
  - Ensure that related actions are performed together, or not at all, by funneling activity for related tables through a single path
- Improved code clarity: By using appropriate identifier names to describe the actions of the routine, you reduce the need for comments and enhance clarity.

Summary

In this lesson, you should have learned that:

- **A function is a named PL/SQL block that must return a value.**
- **A function is created by using the `CREATE FUNCTION` syntax.**
- **A function is invoked as part of an expression.**
- **A function stored in the database can be called in SQL statements.**
- **A function can be removed from the database by using the `DROP FUNCTION` syntax.**
- **Generally, you use a procedure to perform an action and a function to compute a value.**

ORACLE

Summary

A function is a named PL/SQL block that must return a value. Generally, you create a function to compute and return a value, and a procedure to perform an action.

A function can be created or dropped.

A function is invoked as a part of an expression.

Practice 10 Overview

This practice covers the following topics:

- **Creating stored functions**
  - **To query a database table and return specific values**
  - **To be used in a SQL statement**
  - **To insert a new row, with specified parameter values, into a database table**
  - **Using default parameter values**
- **Invoking a stored function from a SQL statement**
- **Invoking a stored function from a stored procedure**

ORACLE

Practice 10 Overview

If you encounter compilation errors when using *i*SQL*Plus, use the `SHOW ERRORS` command.

If you correct any compilation errors in *i*SQL*Plus, do so in the original script file, not in the buffer, and then rerun the new version of the file. This will save a new version of the program unit to the data dictionary.

Practice 10

1. Create and invoke the Q_JOB function to return a job title.
   a. Create a function called Q_JOB to return a job title to a host variable.
   b. Compile the code; create a host variable G_TITLE and invoke the function with job ID SA_REP. Query the host variable to view the result.

G_TITLE
Sales Representative

2. Create a function called ANNUAL_COMP to return the annual salary by accepting two parameters: an employee's monthly salary and commission. The function should address NULL values.
   a. Create and invoke the function ANNUAL_COMP, passing in values for monthly salary and commission. Either or both values passed can be NULL, but the function should still return an annual salary, which is not NULL. The annual salary is defined by the basic formula:

      ```
      (salary*12) + (commission_pct*salary*12)
      ```

   b. Use the function in a SELECT statement against the EMPLOYEES table for department 80.

EMPLOYEE_ID	LAST_NAME	Annual Compensation
145	Russell	235200
146	Partners	210600
147	Errazuriz	187200
148	[illegible]	[illegible]
177	Livingston	120960
179	Johnson	81840

34 rows selected.

3. Create a procedure, NEW_EMP, to insert a new employee into the EMPLOYEES table. The procedure should contain a call to the VALID_DEPTID function to check whether the department ID specified for the new employee exists in the DEPARTMENTS table.
   a. Create the function VALID_DEPTID to validate a specified department ID. The function should return a BOOLEAN value.
   b. Create the procedure NEW_EMP to add an employee to the EMPLOYEES table. A new row should be added to the EMPLOYEES table if the function returns TRUE. If the function returns FALSE, the procedure should alert the user with an appropriate message.

      Define default values for most parameters. The default commission is 0, the default salary is 1000, the default department number is 30, the default job is SA_REP, and the default manager number is 145. For the employee's ID number, use the sequence EMPLOYEES_SEQ. Provide the last name, first name, and e-mail address for the employee.
   c. Test your NEW_EMP procedure by adding a new employee named Jane Harris to department 15. Allow all other parameters to default. What was the result?
   d. Test your NEW_EMP procedure by adding a new employee named Joe Harris to department 80. Allow all other parameters to default. What was the result?

11

Managing Subprograms

Objectives

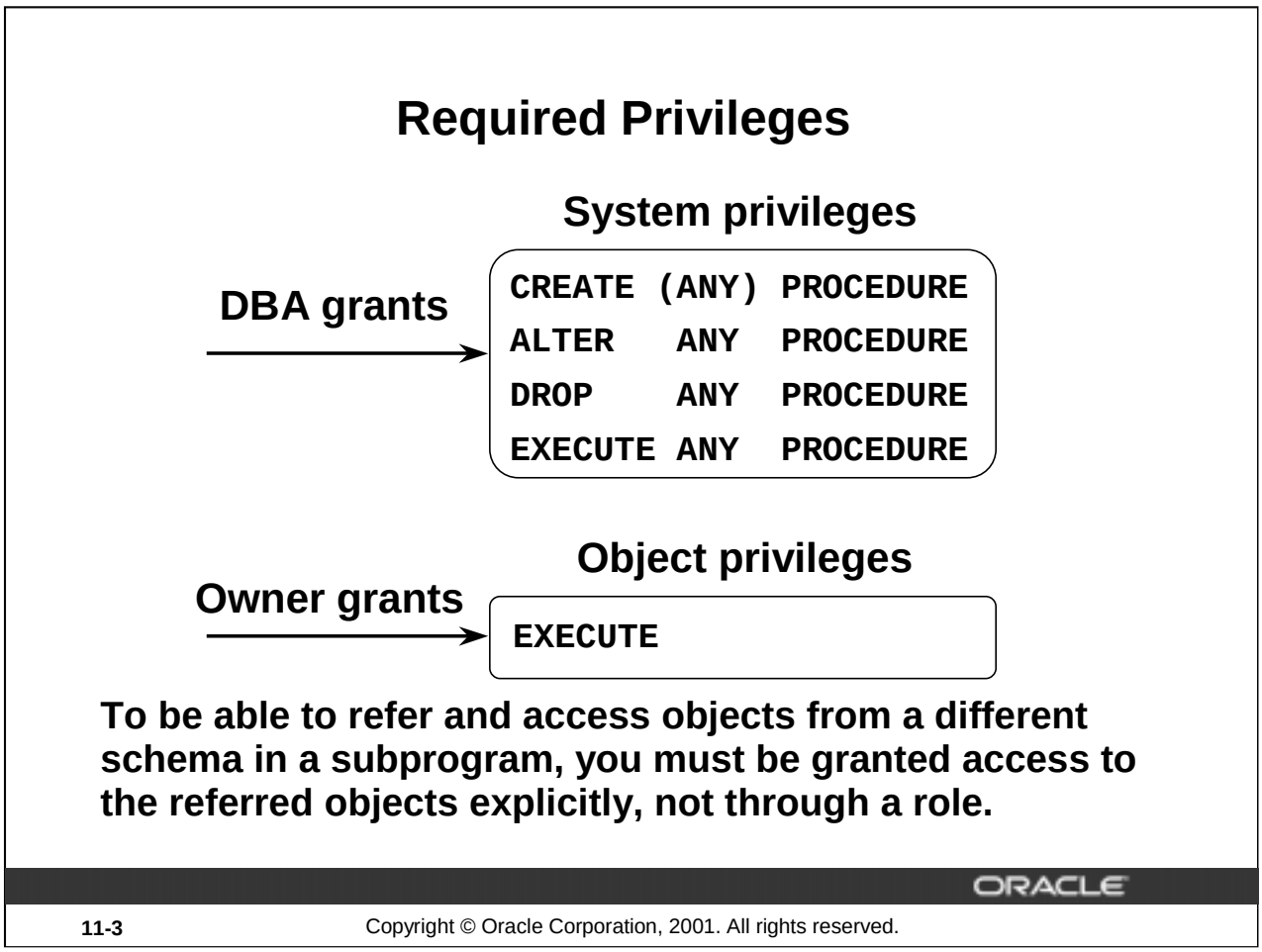
After completing this lesson, you should be able to do the following:

- **Contrast system privileges with object privileges**
- **Contrast invokers rights with definers rights**
- **Identify views in the data dictionary to manage stored objects**
- **Describe how to debug subprograms by using the `DBMS_OUTPUT` package**

ORACLE

Lesson Aim

This lesson introduces you to system and object privilege requirements. You learn how to use the data dictionary to gain information about stored objects. You also learn how to debug subprograms.



System and Object Privileges

There are more than 80 system privileges. Privileges that use the word CREATE or ANY are system privileges; for example, `GRANT ALTER ANY TABLE TO green;`. System privileges are assigned by user SYSTEM or SYS.

Object privileges are rights assigned to a specific object within a schema and always include the name of the object. For example, Scott can assign privileges to Green to alter his EMPLOYEES table as follows:

```
GRANT ALTER ON employees TO green;
```

To create a PL/SQL subprogram, you must have the system privilege CREATE PROCEDURE. You can alter, drop, or execute PL/SQL subprogram without any further privileges being required.

If a PL/SQL subprogram refers to any objects that are not in the same schema, you must be granted access to these explicitly, not through a role.

If the ANY keyword is used, you can create, alter, drop, or execute your own subprograms and those in another schema. Note that the ANY keyword is optional only for the CREATE PROCEDURE privilege.

You must have the EXECUTE object privilege to invoke the PL/SQL subprogram if you are not the owner and do not have the EXECUTE ANY system privilege.

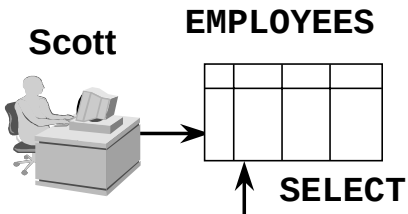
By default the PL/SQL subprogram executes under the security domain of the owner.

Note: The keyword PROCEDURE is used for stored procedures, functions, and packages.

Granting Access to Data

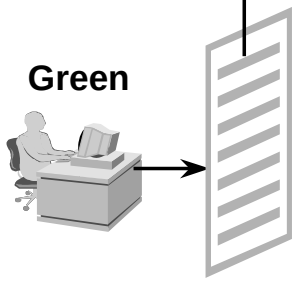
Direct access:

```
GRANT SELECT
 ON   employees
 TO   scott;
Grant Succeeded.
```



Indirect access:

```
GRANT EXECUTE
 ON   query_emp
 TO   green;
Grant Succeeded.
```



The procedure executes with the privileges of the owner (default).

ORACLE

Providing Indirect Access to Data

Suppose the EMPLOYEES table is located within the PERSONNEL schema, and there is a developer named Scott and an end user named Green. Ensure that Green can access the EMPLOYEES table only by way of the QUERY_EMP procedure that Scott created, which queries employee records.

Direct Access

- From the PERSONNEL schema, provide object privileges on the EMPLOYEES table to Scott.
- Scott creates the QUERY_EMP procedure that queries the EMPLOYEES table.

Indirect Access

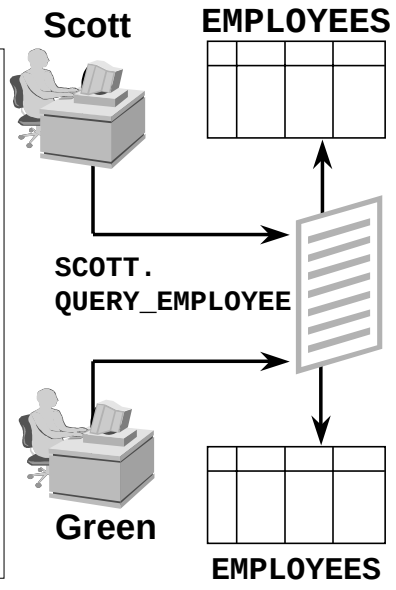
Scott provides the EXECUTE object privilege to Green on the QUERY_EMP procedure.

By default the PL/SQL subprogram executes under the security domain of the owner. This is referred to as definer's-rights. Because Scott has direct privileges to EMPLOYEES and has created a procedure called QUERY_EMP, Green can retrieve information from the EMPLOYEES table by using the QUERY_EMP procedure.

Using Invoker's-Rights

The procedure executes with the privileges of the user.

```
CREATE PROCEDURE query_employee
(p_id IN employees.employee_id%TYPE,
 p_name OUT employees.last_name%TYPE,
 p_salary OUT employees.salary%TYPE,
 p_comm OUT
   employees.commission_pct%TYPE)
AUTHID CURRENT_USER
IS
BEGIN
  SELECT last_name, salary,
         commission_pct
   INTO  p_name, p_salary, p_comm
   FROM  employees
   WHERE employee_id=p_id;
END query_employee;
```



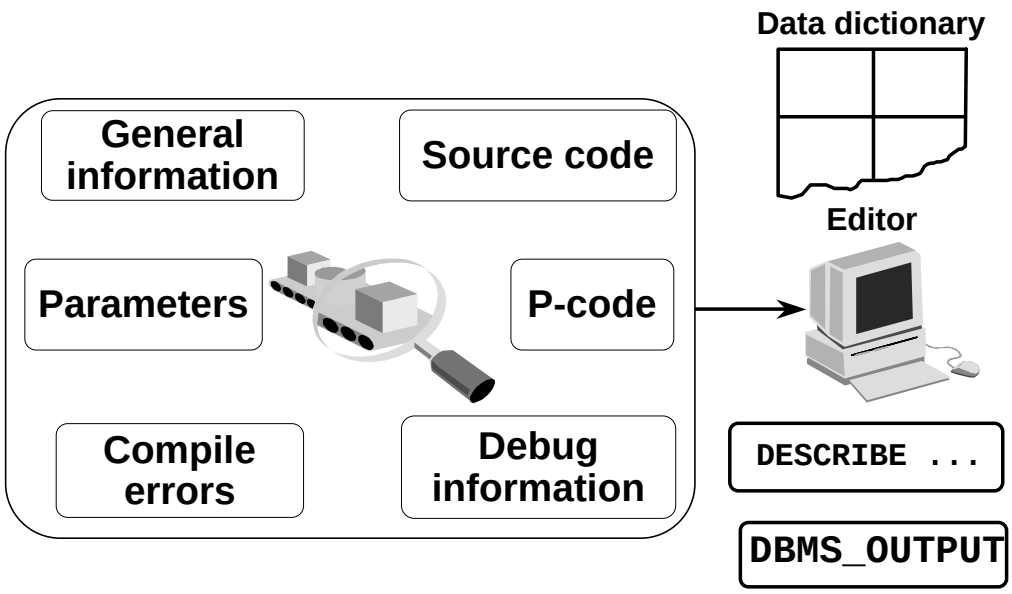
ORACLE

Invoker's-Rights

To ensure that the procedure executes using the security of the executing user, and not the owner, use `AUTHID CURRENT_USER`. This ensures that the procedure executes with the privileges and schema context of its current user.

Default behavior, as shown on the previous page, is when the procedure executes under the security domain of the owner; but if you wanted to explicitly state that the procedure should execute using the owner's privileges, then use `AUTHID DEFINER`.

Managing Stored PL/SQL Objects



ORACLE

Stored Information	Description	Access Method
General	Object information	The `USER_OBJECTS` data dictionary view
Source code	Text of the procedure	The `USER_SOURCE` data dictionary view
Parameters	Mode: `IN`/ `OUT`/ `IN OUT`, datatype	*i*SQL*Plus: `DESCRIBE` command
P-code	Compiled object code	Not accessible
Compile errors	PL/SQL syntax errors	The `USER_ERRORS` data dictionary view *i*SQL*Plus: `SHOW ERRORS` command
Run-time debug information	User-specified debug variables and messages	The `DBMS_OUTPUT` Oracle-supplied package

USER_OBJECTS

Column	Column Description
OBJECT_NAME	Name of the object
OBJECT_ID	Internal identifier for the object
OBJECT_TYPE	Type of object, for example, TABLE, PROCEDURE, FUNCTION, PACKAGE, PACKAGE BODY, TRIGGER
CREATED	Date when the object was created
LAST_DDL_TIME	Date when the object was last modified
TIMESTAMP	Date and time when the object was last recompiled
STATUS	VALID or INVALID

*Abridged column list

ORACLE

Using USER_OBJECTS

To obtain the names of all PL/SQL stored objects within a schema, query the USER_OBJECTS data dictionary view.

You can also examine the ALL_OBJECTS and DBA_OBJECTS views, each of which contains the additional OWNER column, for the owner of the object.

List All Procedures and Functions

```
SELECT object_name, object_type
FROM  user_objects
WHERE object_type in ('PROCEDURE',
'FUNCTION')ORDER BY object_name;
```

OBJECT_NAME	OBJECT_TYPE
ADD_DEPT	PROCEDURE
ADD_JOB	PROCEDURE
ADD_JOB_HISTORY	PROCEDURE
ANNUAL_COMP	FUNCTION
DEL_JOB	PROCEDURE
FORMAT_PHONE	PROCEDURE
LEAVE_EMP	PROCEDURE
LEAVE_EMP2	PROCEDURE
LOC[illegible]ITION	PROCEDURE

20 rows selected.

ORACLE

Example

The example in the slide displays the names of all the procedures and functions that you have created.

USER_SOURCE Data Dictionary View

Column	Column Description
NAME	Name of the object
TYPE	Type of object, for example, PROCEDURE, FUNCTION, PACKAGE, PACKAGE BODY
LINE	Line number of the source code
TEXT	Text of the source code line

ORACLE

Using USER_SOURCE

To obtain the text of a stored procedure or function, use the USER_SOURCE data dictionary view.

Also examine the ALL_SOURCE and DBA_SOURCE views, each of which contains the additional OWNER column, for the owner of the object.

If the source file is unavailable, you can use SQL*Plus to regenerate it from USER_SOURCE.

List the Code of Procedures and Functions

```
SELECT  text
FROM  user_source
WHERE  name = 'QUERY_EMPLOYEE'
ORDER BY  line;
```

TEXT
PROCEDURE query_employee
(p_id IN employees.employee_id%TYPE, p_name OUT employees.last_name%TYPE,
p_salary OUT employees.salary%TYPE, p_comm OUT employees.commission_pct%TYPE)
AUTHID CURRENT_USER
IS
BEGIN
SELECT last_name, salary, commission_pct
INTO p_name,p_salary,p_comm
FROM employees
WHERE employee_id=p_id;
END query_employee;

11 rows selected.

ORACLE

Example

Use the USER_SOURCE data dictionary view to display the complete text for the QUERY_EMPLOYEE procedure.

USER_ERRORS

Column	Column Description
NAME	Name of the object
TYPE	Type of object, for example, PROCEDURE, FUNCTION, PACKAGE, PACKAGE BODY, TRIGGER
SEQUENCE	Sequence number, for ordering
LINE	Line number of the source code at which the error occurs
POSITION	Position in the line at which the error occurs
TEXT	Text of the error message

ORACLE

Obtaining Compile Errors

To obtain the text for compile errors, use the USER_ERRORS data dictionary view or the SHOW ERRORS *i*SQL*Plus command.

Also examine the ALL_ ERRORS and DBA_ ERRORS views, each of which contains the additional OWNER column, for the owner of the object.

Detecting Compilation Errors: Example

```
CREATE OR REPLACE PROCEDURE log_execution
IS
BEGIN
INPUT INTO log_table (user_id, log_date)
                                    -- wrong
VALUES (USER, SYSDATE);
END;
/
```

```
Warning: Procedure created with compilation errors.
```

ORACLE

Example

Given the above code for `LOG_EXECUTION`, there will be a compile error when you run the script for compilation.

List Compilation Errors by Using USER_ERRORS

```
SELECT line || '/' || position POS, text
FROM   user_errors
WHERE  name = 'LOG_EXECUTION'
ORDER BY line;
```

POS	TEXT
4/7	PLS-00103: Encountered the symbol "INTO" when expecting one of the following: := . (@ % ;
5/1	PLS-00103: Encountered the symbol "VALUES" when expecting one of the following: . (, % ; limit The symbol "VALUES" was ignored.
6/1	PLS-00103: Encountered the symbol "END"

ORACLE

Listing Compilation Errors, Using USER_ERRORS

The SQL statement above is a SELECT statement from the USER_ERRORS data dictionary view, which you use to see compilation errors.

List Compilation Errors by Using SHOW ERRORS

```
SHOW ERRORS PROCEDURE log_execution
```

Errors for PROCEDURE LOG_EXECUTION

LINE/COL	ERROR
4/7	PLS-00103: Encountered the symbol "INTO" when expecting one of th e following: := . (@ % ;
5/1	PLS-00103: Encountered the symbol "VALUES" when expecting one of the following: . (, % ; limit The symbol "VALUES" was ignore d.
6/1	PLS-00103: Encountered the symbol "END"

ORACLE

SHOW ERRORS

Use SHOW ERRORS without any arguments at the SQL prompt to obtain compilation errors for the last object you compiled.

You can also use the command with a specific program unit. The syntax is as follows:

```
SHOW ERRORS [{FUNCTION|PROCEDURE|PACKAGE|PACKAGE
              BODY|TRIGGER|VIEW} [schema.]name]
```

Using the SHOW ERRORS command, you can view only the compilation errors that are generated by the latest statement that is used to create a subprogram. The USER_ERRORS data dictionary view stores all the compilation errors generated previously while creating subprograms.

DESCRIBE in *i*SQL*Plus

```
DESCRIBE query_employee
DESCRIBE add_dept
DESCRIBE tax
```

PROCEDURE **query_employee**

Argument Name	Type	In/Out	Default?
P_ID	NUMBER(6)	IN	
P_NAME	VARCHAR2(25)	OUT	
P_SALARY	NUMBER(8,2)	OUT	
P_COMM	NUMBER(2,2)	OUT	

PROCEDURE add_dept

Argument Name	Type	In/Out	Default?
P_NAME	VARCHAR2(30)	IN	DEFAULT
P_LOC	NUMBER(4)	IN	DEFAULT

FUNCTION tax RETURNS NUMBER

Argument Name	Type	In/Out	Default?
P_VALUE	NUMBER	IN	

ORACLE

Describing Procedures and Functions

To display a procedure or function and its parameter list, use the *i*SQL*Plus `DESCRIBE` command.

Example

The code in the slide displays the parameter list for the `QUERY_EMPLOYEE` and `ADD_DEPT` procedures and the `TAX` function.

Consider the displayed parameter list for the `ADD_DEPT` procedure, which has defaults. The `DEFAULT` column indicates only that there is a default value; it does not give the actual value itself.

Debugging PL/SQL Program Units

- The `DBMS_OUTPUT` package:
  - Accumulates information into a buffer
  - Allows retrieval of the information from the buffer
- Autonomous procedure calls (for example, writing the output to a log table)
- Software that uses `DBMS_DEBUG`
  - Procedure Builder
  - Third-party debugging software

ORACLE

Debugging PL/SQL Program Units

Different packages that can be used for debugging PL/SQL program units are shown in the slide.

You can use `DBMS_OUTPUT` packaged procedures to output values and messages from a PL/SQL block. This is done by accumulating information into a buffer and then allowing the retrieval of the information from the buffer. `DBMS_OUTPUT` is an Oracle-supplied package. You qualify every reference to these procedures with the `DBMS_OUTPUT` prefix.

Benefits of Using `DBMS_OUTPUT` Package

This package enables developers to follow closely the execution of a function or procedure by sending messages and values to the output buffer. Within *i*SQL*Plus use `SET SERVEROUTPUT ON` or `OFF` instead of using the `ENABLE` or `DISABLE` procedure.

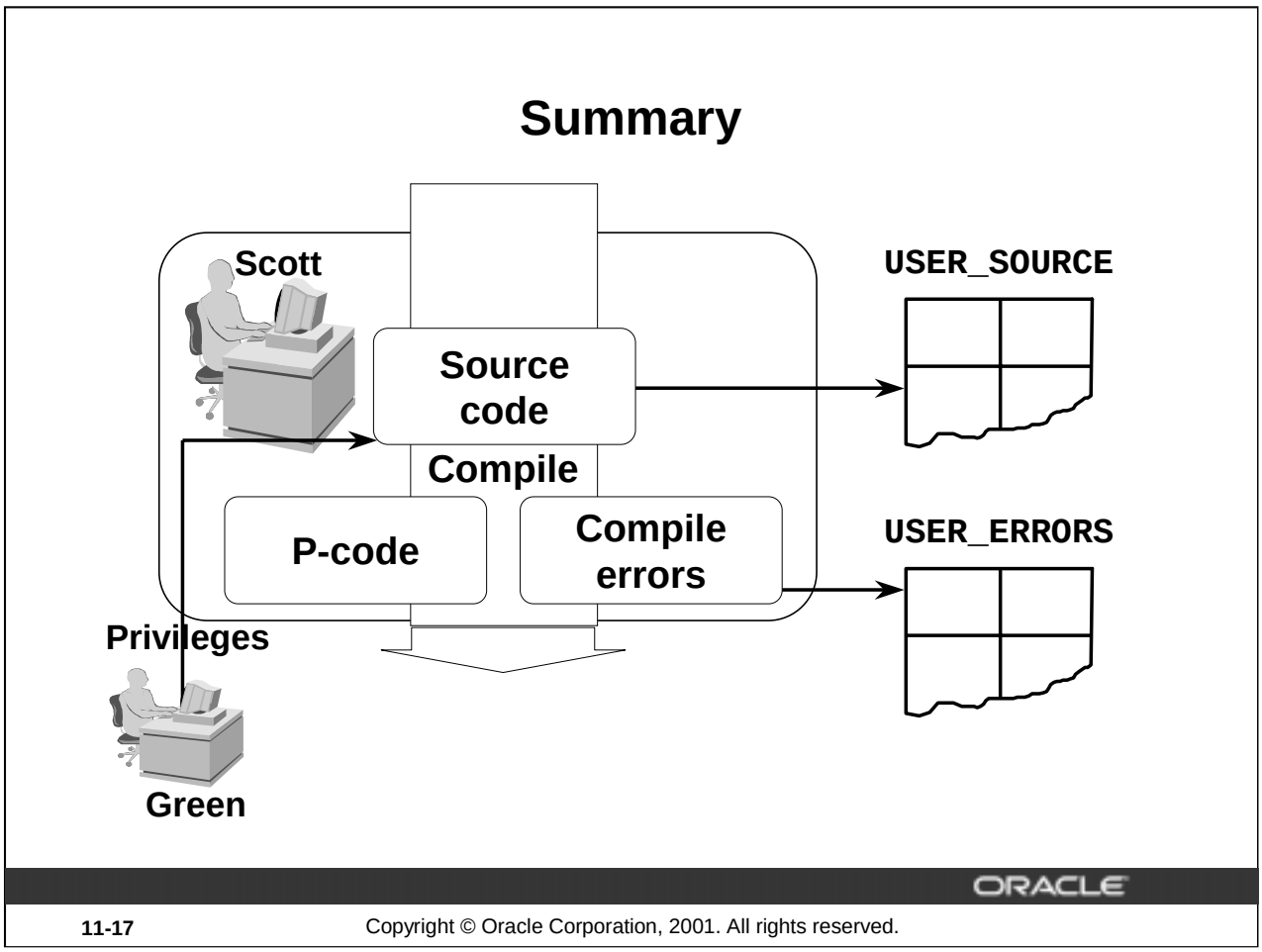
Suggested Diagnostic Information

- Message upon entering, leaving a procedure, or indicating that an operation has occurred
- Counter for a loop
- Value for a variable before and after an assignment

Note: The buffer is not emptied until the block terminates.

You can debug subprograms by specifying autonomous procedure calls and store the output as values of columns into a log table.

Debugging using Oracle Procedure Builder is discussed in Appendix C. Procedure Builder uses a Oracle-specified debugging package called `DBMS_DEBUG`.



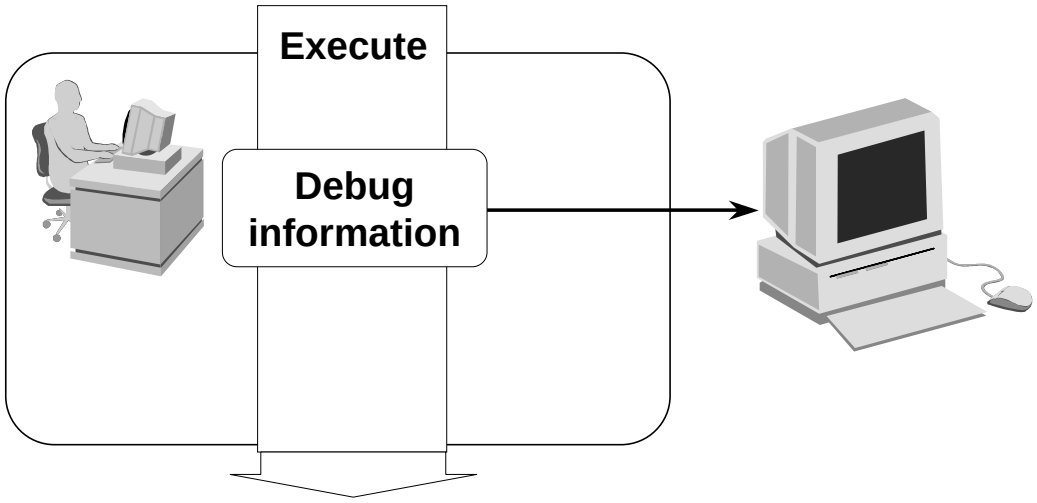
Summary

A user must be granted the necessary privileges to access database objects through a subprogram.

Take advantage of various data dictionary views, SQL commands, *i*SQL*Plus commands, and Oracle-supplied procedures to manage a stored procedure or function during its development cycle.

Name	Data Dictionary View or Command	Description
USER_OBJECTS	Data dictionary view	Provides general information about the object
USER_SOURCE	Data dictionary view	Provides the text of the object, (that is, the PL/SQL block)
DESCRIBE	*i*SQL*Plus command	Provides the declaration of the object
USER_ERRORS	Data dictionary view	Shows compilation errors
SHOW ERRORS	*i*SQL*Plus command	Shows compilation errors, per procedure or function
DBMS_OUTPUT	Oracle-supplied package	Provides user-specified debugging, giving variable values and messages
GRANT	*i*SQL command	Provides the security privileges for the owner who creates the procedure and the user who runs it, enabling them to perform their respective operations

Summary



Summary (continued)

- Query the data dictionary.
  - List all your procedures and functions, using the USER_OBJECTS view.
  - List the text of certain procedures or functions, using the USER_SOURCE view.
- Prepare a procedure: Recreate a procedure and display any compile errors automatically.
- Test a procedure: Test a procedure by supplying input values; test a procedure or function by displaying output or return values.

Practice 11 Overview

This practice covers the following topics:

- **Recreating the source file for a procedure**
- **Recreating the source file for a function**

ORACLE

Practice 11 Overview

In this practice you will recreate the source code for a procedure and a function.

Practice 11

Suppose you have lost the code for the `NEW_EMP` procedure and the `VALID_DEPTNO` function that you created in lesson 10. (If you did not complete the practices in lesson 10, you can run the solution scripts to create the procedure and function.)

Create a *i*SQL*Plus spool file to query the appropriate data dictionary view to regenerate the code.

Hint:

```
SET             -- options ON|OFF
SELECT          -- statement(s) to extract the code
SET             -- reset options ON|OFF
```

To spool the output of the file to a `.sql` file from *i*SQL*Plus, select the `Save` option for the `Output` and execute the code.