Oracle PL/SQL Training

with examples and
hands-on exercises

WEBUCATOR

Copyright © 2022 by Webucator. All rights reserved.

No part of this manual may be reproduced or used in any manner without written permission of the copyright owner.

Version: 1.3.3

The Authors

Nat Dunn

Nat Dunn is the founder of Webucator (www.webucator.com), a company that has provided training for tens of thousands of students from thousands of organizations. Nat started the company in 2003 to combine his passion for technical training with his business expertise, and to help companies benefit from both. His previous experience was in sales, business and technical training, and management. Nat has an MBA from Harvard Business School and a BA in International Relations from Pomona College.

Follow Nat on Twitter at @natdunn and Webucator at @webucator.

Stephen Withrow (Editor)

Stephen has over 30 years of experience in training, development, and consulting in a variety of technology areas including Python, Java, C, C++, XML, JavaScript, Tomcat, JBoss, Oracle, and DB2. His background includes design and implementation of business solutions on client/server, Web, and enterprise platforms. Stephen has a degree in Computer Science and Physics from Florida State University.

Class Files

Download the class files used in this manual at https://static.webucator.com/media/public/materials/classfiles/ORA200-1.3.3.zip.

Errata

Corrections to errors in the manual can be found at https://www.webucator.com/books/errata/.

Table of Contents

LESSON 1
PL/SQL Basics

Topics Covered

☑ What us PL/SQL is?

☑ Variables.

☑ PL/SQL data types.

☑ Main features of PL/SQL.

☑ PL/SQL code blocks.

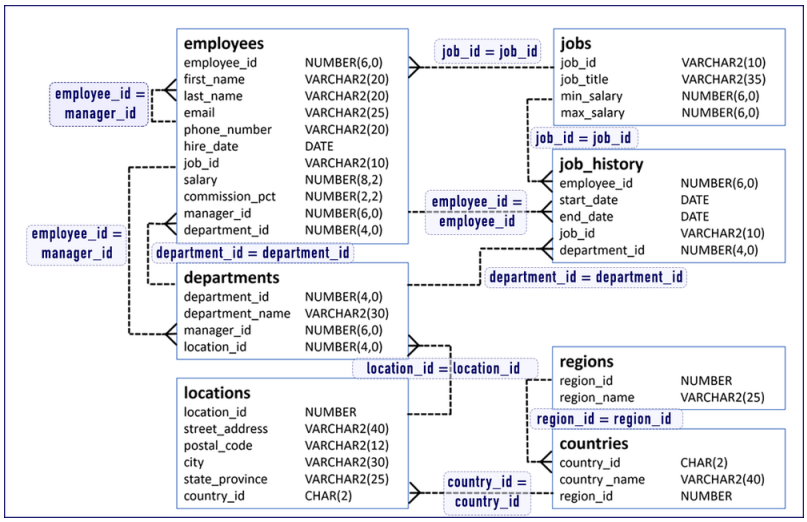
Evaluation Copy

Introduction

In this lesson, we provide a quick introduction to the features of PL/SQL.

1.1. The HR Schema

In setting up, you should have created the HR user and then run a couple of scripts to create and populate schema objects for a fictional Human Resources company. If you haven't completed your setup, you should do so now. A simple entity relationship diagram showing the relationships between the tables in the **HR** schema is shown below:



This diagram is included in a PDF called `hr-entity-diagram.pdf` in your class files. We recommend you print that out to have as a reference.

If at any point, you want to reset the **HR** schema back to its original structure and content, you can do so by opening and executing the `reset-hr-schema.sql` file in your class files.

Evaluation Copy

1.2. What is PL/SQL?

PL/SQL stands for Procedural Language/Structured Query Language. It is an Oracle-specific extension to SQL that allows developers to declare variables, manage flow control of an application, create subprograms, and catch runtime errors. Through creating procedures and functions, developers create modular, reusable code for performing complex tasks. These programs can be called explicitly or prompted by an event (a trigger). PL/SQL code can be packaged for organizational purposes and for reuse and sharing of applications.

In these lessons, you will learn to use the following features of PL/SQL:

1. Subprograms
2. Variables and Constants
3. Conditional Processing
4. Exceptions
5. Loops
6. Cursors

EVALUATION COPY: Not to be used in class.

7. Packages
8. Triggers

1.3. Blocks

PL/SQL code is written in blocks, which are divided into three parts:

1. `DECLARE` - contains declarations of variables, constants, cursors, subprograms, etc.
2. `BEGIN` - contains the executable commands.
3. `EXCEPTION` - contains error handling code.

The syntax is as follows:

```
DECLARE
  /* declarations */
BEGIN
  /* executable commands */
EXCEPTION
  /* error handling code */
END;
```

Evaluation Copy

The block must end with the `END` statement, but only the `BEGIN` part of the block is required, so the simplest valid block looks like this:

```
BEGIN
  /* executable commands */
END;
```

❖ 1.3.1. Comments

1. Single-line comments begin with two dashes (`--`).
2. Multi-line comments begin with `/*` and end with `*/`.

```
-- This is a single-line comment.

/*
  This is
  a multi-line
  comment.
*/
```

1.4. Outputting Information

The built-in `DBMS_OUTPUT` (for **D**ata**B**ase **M**anagement **S**ystem **Output**) package includes a `PUT_LINE()` procedure that is useful for outputting information. It's great for testing and debugging. We will use it a lot to get feedback on our code.

It is tradition when teaching a new programming language to start with a "Hello, world!" script, so here is ours:

Demo 1.1: PL-SQL-Basics/Demos/hello-world.sql

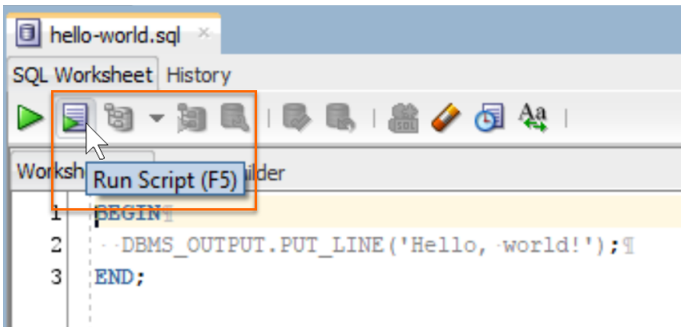
Evaluation Copy

```
1.  BEGIN
2.    DBMS_OUTPUT.PUT_LINE('Hello, world!');
3.  END;
```

Open this file in SQL Developer and run it.

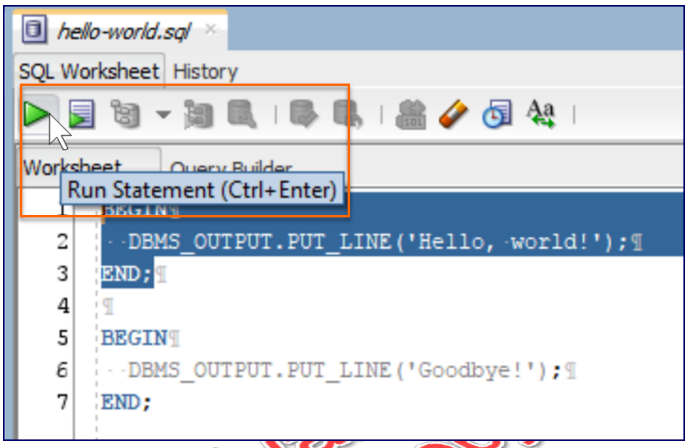
Running Code in SQL Developer

When you have a SQL file open in SQL Developer, you can run the whole file by pressing **F5** or clicking the **Run Script** icon:

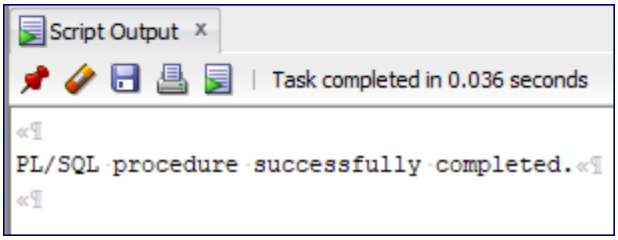


EVALUATION COPY: Not to be used in class.

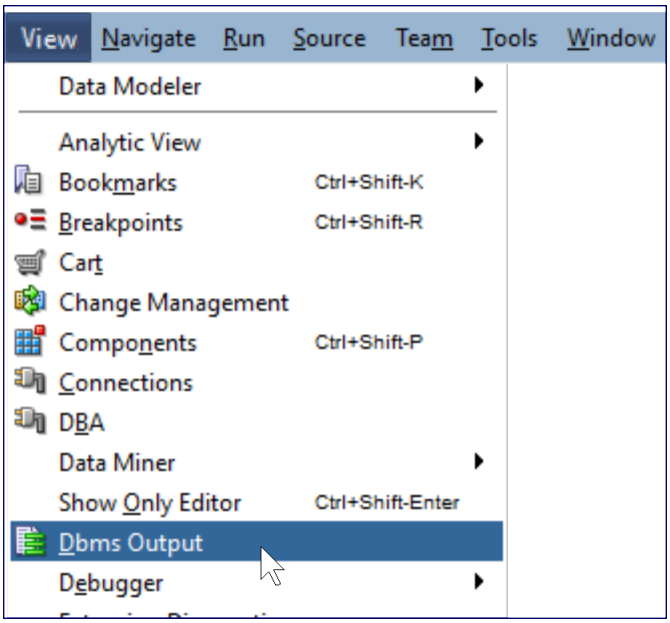
To run a portion of the file, highlight the portion you want to run and press **Ctrl+Enter** or click the **Run Statement** icon:



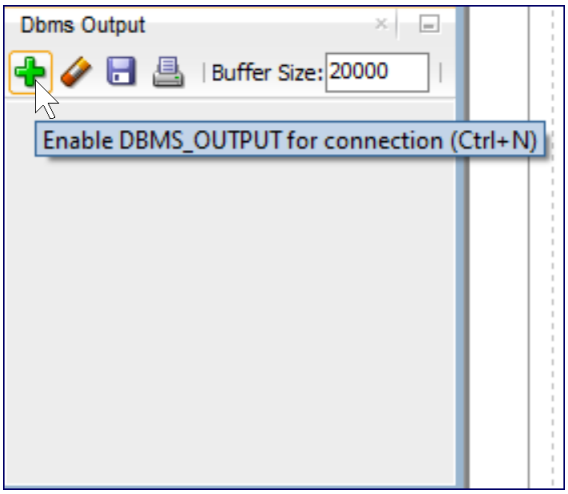
You'll most likely see a **Script Output** window open up with the following output:



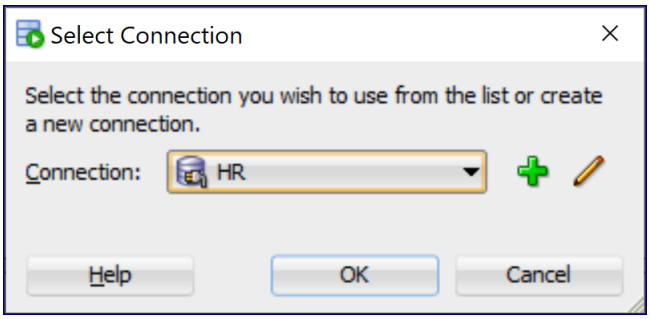
You'll notice that it does not output "Hello, world!" That's because that output is disabled by default. However, SQL Developer includes a **Dbms Output** window that you open up via the **View** menu:



You then must make your connection by clicking the green plus symbol in the **Dbms Output** window:

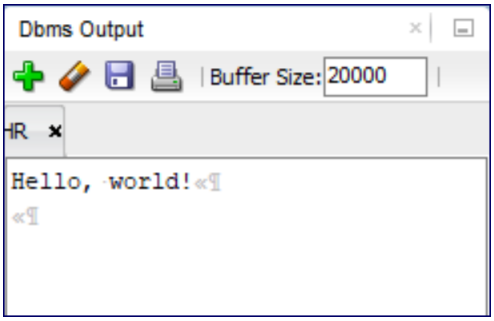


Select the **HR** connection:



EVALUATION COPY: Not to be used in class.

And now run the Hello-world code again. This time you should see the following output in the **Dbms Output** window:



Enabling Output

It's also easy to enable output in the **Script Output** window. Just run the following command:

```
SET SERVEROUTPUT ON
```

See this YouTube video (https://youtu.be/u3g7p3aWNOg) for instructions on how to enable this by default in SQL Server Developer.

Evaluation Copy

1.5. Variables and Constants

Variables and constants are mechanisms for storing data in memory. As their names suggest, values stored in variables can change (they are *variable*), but values stored in constants cannot change (they are *constant*).

❖ 1.5.1. Declaring Variables

All of the examples in this section are in the `PL-SQL-Basics/Demos/variables.sql` file. Please open that file in SQL Developer so you can run the code shown to see the results.

Variables are declared in the `DECLARE` part of a PL/SQL block:

```
DECLARE
  variable_name datatype := default_value;
BEGIN
  /* executable commands */
END;
```

Variables are declared as a specific data type. When they are declared, they can also be *initialized*, meaning that they can be assigned an initial value. The operator for assigning values to variables is `:=` (a colon followed by an equals sign).

Here is our "Hello, world!" code using a variable:

```
DECLARE
  greeting VARCHAR2(20) := 'Hello, world!';
BEGIN
  DBMS_OUTPUT.PUT_LINE(greeting);
END;
```

Evaluation Copy

And here is the same code, but after outputting the variable the first time, we change its value and output it again:

```
DECLARE
  -- Declare and initialize variable
  greeting VARCHAR2(20) := 'Hello, world!';
BEGIN
  DBMS_OUTPUT.PUT_LINE(greeting);

  -- Assign a new value
  greeting := 'Hello again!';

  DBMS_OUTPUT.PUT_LINE(greeting);
END;
```

As we mentioned, variables can be initialized when they are declared. But they don't have to be. In the code below, we don't assign an initial value to `greeting`. Its initial value will therefore be `NULL`. We then assign it a value before outputting it:

EVALUATION COPY: Not to be used in class.

```
DECLARE
  -- Declare, but don't initialize variable
  greeting VARCHAR2(20);
BEGIN
  -- Assign value
  greeting := 'Hello, world';
  DBMS_OUTPUT.PUT_LINE(greeting);
END;
```

❖ 1.5.2. Default Values

When you initialize a variable on declaration, you are essentially assigning a default value to that variable. Another way to do the same thing is to use the `DEFAULT` keyword. The following two statements are equivalent:

Evaluation Copy

```
greeting VARCHAR2(20) := 'Hello, world!';
greeting VARCHAR2(20) DEFAULT 'Hello, world!';
```

You can use the two interchangeably. Throughout these lessons, we will usually use the assignment operator.

❖ 1.5.3. Concatenation

Concatenation is a fancy word for stringing together different words or characters. In Oracle, the concatenation operator is the double pipe (`||`).

Often, you will concatenate variables with literal values, as the example below shows:

```
DECLARE
  greeting VARCHAR2(20);
  person   VARCHAR2(20);
  subject  VARCHAR2(20) := 'PL/SQL';
BEGIN
  greeting := 'Hello';
  person   := 'Dolly';
  DBMS_OUTPUT.PUT_LINE(greeting || ', ' || person || '!');

  -- You can break long lines up:
  DBMS_OUTPUT.PUT_LINE(greeting || ', ' || person ||
    '! You are learning ' || subject ||
    ' really fast. Great job!');
END;
```

The output of this code will be:

```
Hello, Dolly!
Hello, Dolly! You are learning PL/SQL really fast. Great job!
```

Evaluation Copy

1.6. Constants

All of the examples in this section are in the `PL-SQL-Basics/Demos/constants.sql` file. Please open that file in SQL Developer so you can run the code shown to see the results.

Constants are similar to variables, but they must be assigned values when they are declared and those values cannot be changed. They are declared with the `CONSTANT` keyword:

```
pi CONSTANT FLOAT := 3.14159;
```

If you attempt to change the value of a constant, a compilation error will occur, meaning the PL/SQL compiler, which is responsible for converting PL/SQL into the low-level language that your computer understands, is unable to make this conversion. To see this, try running the following code:

EVALUATION COPY: Not to be used in class.

```
DECLARE
  pi CONSTANT FLOAT := 3.14159;
BEGIN
  pi := 3.1415926535897932;
END;
```

This will result in the following error:

```
PLS-00363: expression 'PI' cannot be used as an assignment target
```

The following code uses a constant and two variables, one of which is initialized on declaration:

```
DECLARE
  pi CONSTANT   FLOAT   := 3.14159;
  radius        NUMBER  := 5;
  area          FLOAT;
BEGIN
  area := pi * POWER(radius, 2);
  DBMS_OUTPUT.PUT_LINE(area);
END;
```

Evaluation Copy

1.7. Data Types

Variables can be of any valid PL/SQL data type, which includes all the SQL data types, the most common of which are shown in the table below:

Common Oracle Data Types

Data Type	Description
`CHAR [(size)]`	Character data with length of `size`, which defaults to 1.
`NCHAR [(size)]`	Unicode character data with length of `size`, which defaults to 1.
`VARCHAR2(size)`	Character data with length of up to `size`.
`NVARCHAR2(size)`	Unicode character data with length of up to `size`.
`LONG`	Character data of variable length up to 2 gigabytes. Used to store large amounts of data.
`NUMBER [(p[,s])]`	A number. The precision (`p`) specifies the total number of digits allowed. The scale (`s`) specifies the number of digits after the decimal point.
`FLOAT [(p)]`	A number with a floating point (i.e., a decimal). The precision (`p`) specifies the total number of digits allowed.
`DATE`	A date.
`TIMESTAMP`	A specific point in time.

Square Brackets in Code Notation

Evaluation Copy

Square brackets in code notation means that the contained portion is optional. To illustrate, consider the following from the table above:

```
NUMBER [(p[,s])]
```

This means that all of the following are valid ways of expressing a number:

1. `NUMBER` - No precision is specified.
2. `NUMBER(6)` - Precision is specified, but scale is not.
3. `NUMBER(6,3)` - Precision and scale are specified.

Note that scale cannot be specified unless precision is also specified. The outside square brackets in `[(p[,s])]` indicate that the whole section is optional. The inside square brackets indicate that scale is optional even if precision is specified. If it were written as `[(p,s)]`, it would indicate that precision and scale are optional, but if one is included, the other must be included as well.

EVALUATION COPY: Not to be used in class.

❖ 1.7.1. Boolean Data Type

One of the most useful data types that is available in PL/SQL, **but not in SQL**, is the `BOOLEAN` data type, which can be one of three possible values:

1. `TRUE`
2. `FALSE`
3. `NULL` (indicating the value is unknown).

We will discuss `BOOLEAN` data types when we learn about PL/SQL conditionals.

See `https://docs.oracle.com/en/database/oracle/oracle-database/19/lnpls/plsql-data-types.html` for complete information on PL/SQL data types.

1.8. Naming Variables and Other Elements

Evaluation Copy

The names of elements, such as variables and constants, are called *identifiers* in PL/SQL. We will encounter many other elements, such as functions, procedures, packages, and exceptions, that must also be named with identifiers. The rules for such identifiers are listed below:

1. Identifiers can include letters, digits, dollar signs (`$`), underscores (`_`), and number signs (`#`), but must start with a letter.
2. Identifiers are case insensitive, meaning that `greeting`, `Greeting` and `GREETING` all identify the same element.
3. You may not use Oracle's reserved words[1] as identifiers.
4. As of Oracle 12.2, identifiers can be up to 128 characters long. Before that, the limit was 30 characters.

Identifiers and Double Quotes

While it is possible to get more flexibility in choosing identifiers by using double quotes, this practice is not recommended. See the documentation on Quoted User-Defined Identifiers[2] for details.

1. https://docs.oracle.com/en/database/oracle/oracle-database/19/lnpls/plsql-reserved-words-keywords.html
2. https://docs.oracle.com/en/database/oracle/oracle-database/19/lnpls/plsql-language-fundamentals.html

❖ 1.8.1. Variable Name Prefixes

Variables are *local* to the block within which they are declared. That means that the variable cannot be accessed outside of those blocks. To make this clear and to avoid naming conflicts with table fields and other objects, it is common practice to prefix local variables with `l_`. That is a lowercase "L" followed by an underscore. For example:

```
DECLARE
  -- Declare, but don't initialize variable
  l_greeting VARCHAR2(20);
BEGIN
  -- Assign value
  l_greeting := 'Hello, world';
  DBMS_OUTPUT.PUT_LINE(l_greeting);
END;
```

Likewise, local constants are commonly prefixed with `c_`.

Evaluation Copy

```
DECLARE
  c_pi CONSTANT FLOAT := 3.14159;
  l_radius NUMBER := 5;
  l_area   FLOAT;
BEGIN
l_area := c_pi * POWER(l_radius, 2);
  DBMS_OUTPUT.PUT_LINE(l_area);
END;
```

1.9. Embedding SQL in PL/SQL

It is possible and common to embed SQL within the PL/SQL code. We will see many examples of this. Consider the following:

EVALUATION COPY: Not to be used in class.

Demo 1.2: PL-SQL-Basics/Demos/embedded-sql.sql

```
1.   DECLARE
2.     l_greeting VARCHAR2(50) := 'Hello, world!';
3.   BEGIN
4.     DBMS_OUTPUT.PUT_LINE(l_greeting);
5.
6.     SELECT 'Hello, ' || region_name || '!'
7.     INTO l_greeting
8.     FROM regions
9.     FETCH FIRST 1 ROW ONLY;
10.
11.    DBMS_OUTPUT.PUT_LINE(l_greeting);
12.  END;
```

Code Explanation

This will output:

```
Hello, world!
Hello, Europe!
```

Evaluation Copy

It uses `SELECT … INTO` to select a value from the database and assign that value to a variable.

1.10. SELECT…INTO and RETURNING…INTO

The `SELECT … INTO` syntax is used to select values from database tables into PL/SQL variables. It is thus a way of combining standard SQL with PL/SQL. This syntax allows for selecting values into multiple variables with one `SELECT` statement. The syntax is shown below:

```
SELECT column_name(s)
INTO pl_sql_variable_name(s)
FROM table_name
WHERE condition(s);
```

The following code selects an employee's first name, last name and email address from the `employees` table and stores the results in PL/SQL variables:

Demo 1.3: PL-SQL-Basics/Demos/select-into.sql

```
1.   DECLARE
2.     l_first_name  VARCHAR(20);
3.     l_last_name   VARCHAR(25);
4.     l_email       VARCHAR(25);
5.   BEGIN
6.     SELECT first_name, last_name, email
7.     INTO l_first_name, l_last_name, l_email
8.     FROM employees
9.     WHERE employee_id = 108;
10.
11.    DBMS_OUTPUT.PUT_LINE(l_first_name || ' ' || l_last_name || ': ' ||
12.                         l_email || '.');
13.  END;
```

Code Explanation

This will output:


```
Nancy Greenberg: NGREENBE.
```

Likewise,the `RETURNING … INTO` syntax is used to assign values returned from `UPDATE`, `INSERT`, and `DELETE` statements to PL/SQL variables. The syntax is:

```
DML_STATEMENT
RETURNING column_name(s) INTO pl_sql_variable_name(s);
```

Both `SELECT … INTO` and `RETURNING … INTO` expect *one and only one* row to be returned by the query. An error will occur if more than one row is returned or if no rows are returned. Later, we will learn how to handle these errors.

The following demo shows how to use both of these to capture an employee's salary before and after it is updated:

EVALUATION COPY: Not to be used in class.

Demo 1.4: PL-SQL-Basics/Demos/into.sql

```
1.   DECLARE
2.     l_old_salary NUMBER(8,2);
3.     l_new_salary NUMBER(8,2);
4.   BEGIN
5.     SELECT salary
6.     INTO l_old_salary
7.     FROM employees
8.     WHERE employee_id = 108;
9.
10.    DBMS_OUTPUT.PUT_LINE('Old salary: ' || l_old_salary);
11.
12.    UPDATE employees
13.    SET salary = salary * 1.1
14.    WHERE employee_id = 108
15.    RETURNING salary INTO l_new_salary;
16.
17.    DBMS_OUTPUT.PUT_LINE('New salary: ' || l_new_salary);
18.  END;
```

Evaluation Copy

Code Explanation

This will output:

```
Old salary: 12008
New salary: 13208.8
```

Execute a `ROLLBACK` to undo the change:

```
ROLLBACK;
```

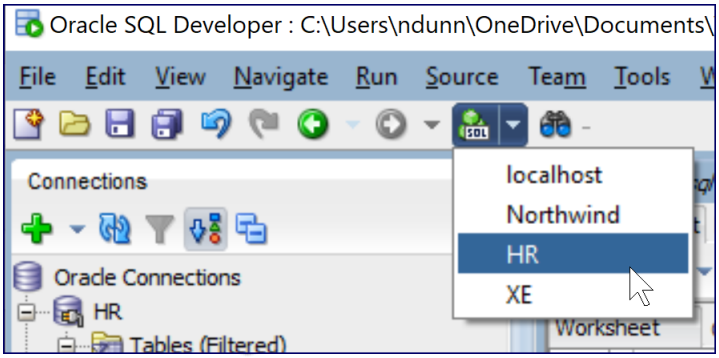
Exercise 1: Using Variables

10 to 15 minutes

In this exercise you will practice writing some basic PL/SQL code.

Opening a New SQL Worksheet in SQL Developer

To open a new SQL worksheet in SQL Developer click the down arrow on the right of the **SQL icon** and select **HR**:



1. Open a new SQL Worksheet in SQL Developer and save it as `block.sql` in `PL-SQL-Basics/Exercises`.
2. Declare three variables:
   A. `l_greeting` with a type of `VARCHAR2(10)` and an initial value of 'Hello'.
   B. `l_first_name` with a type of `VARCHAR2(20)` and no initial value.
   C. `l_last_name` with a type of `VARCHAR2(25)` and no initial value.
3. Select the first and last name of the employee with the job id of "AD_PRES" into the `l_first_name` and `l_last_name` variables.
4. Output a greeting by concatenating the three variables together with some literal strings, so that you get "Hello, Steven King!" The Oracle concatenation operator is a double pipe: `||`.

EVALUATION COPY: Not to be used in class.

Solution: PL-SQL-Basics/Solutions/block.sql

```
1.   DECLARE
2.     l_greeting VARCHAR2(10) := 'Hello';
3.     l_first_name VARCHAR2(20);
4.     l_last_name VARCHAR2(25);
5.   BEGIN
6.     SELECT first_name, last_name
7.     INTO l_first_name, l_last_name
8.     FROM employees
9.     WHERE job_id = 'AD_PRES';
10.
11.    DBMS_OUTPUT.PUT_LINE(l_greeting || ', ' ||
12.      l_first_name || ' ' || l_last_name || '!');
13.  END;
```

1.11. PL/SQL Features

Evaluation Copy

Here is a quick overview of the PL/SQL features we will be covering:

❖ 1.11.1. Subprograms

A subprogram is a named block of PL/SQL code that is stored in a schema and can be called and passed parameters. There are two types of subprograms: procedures and functions. Functions return a result.

❖ 1.11.2. Conditional Processing

A key component to any procedural programming language is the ability to branch your code depending on conditions. Oracle PL/SQL includes `IF-THEN` and `CASE` statements for conditional processing. You will learn about both.

❖ 1.11.3. Exceptions

If an error occurs in your code, PL/SQL will raise an exception, which stops normal execution and transfers control to the `EXCEPTION` part of the block. You will learn how to handle these exceptions.

EVALUATION COPY: Not to be used in class.

❖ 1.11.4. Loops

PL/SQL includes loops, which make it possible to iterate over the same statements using different values. You will learn how to use several types of loops.

❖ 1.11.5. Cursors

A cursor can be used to retrieve rows in a SQL query result set one at a time and provide information about its own state (e.g, whether it has been opened, how many results it has processed, etc.). There are both *implicit* cursors (cursors created every time a SQL `SELECT` or DML statement is executed) and *explicit* cursors (cursors you create to control processing of a SQL `SELECT` or DML statement). You will learn about both types.

❖ 1.11.6. Packages

Evaluation Copy

A package is like an application for storing related PL/SQL code in a database where it can be accessed and reused by other code and applications. You will learn how to create packages.

❖ 1.11.7. Triggers

A trigger is a named piece of code that is fired off when an event occurs in the database. For example, you could use a trigger to perform validation before deleting, updating, or inserting a record into a table. You will learn how to create triggers.

Conclusion

In this lesson, you have had a high-level overview of PL/SQL and learned about variables and data types. Now it's time to start coding.

EVALUATION COPY: Not to be used in class.

LESSON 2
Subprograms

Topics Covered

☑ What is a subprogram?

☑ Procedures.

☑ Functions.

Introduction

A subprogram is a named block of PL/SQL code that is stored in a schema and can be called and passed parameters. There are two types of subprograms: procedures and functions. Functions return a result.

Evaluation Copy

2.1. Introduction to Subprograms

There are many reasons why you would write a subprogram rather than just writing your code in a block and saving it in a file.

1. **Modularity and Maintainability** - By breaking up an application into subprograms that perform specific tasks, you make the application more modular, which makes it more manageable and maintainable.
2. **Reusability** - Subprograms can be called (*invoked*) from other code and can behave differently based on parameters sent to them. This means that the same code can be reused (rather than rewritten!) over and over again. Subprograms can also be used in packages, which we learn about in a later lesson (see page 157).
3. **Performance** - Oracle compiles, stores, and caches subprograms in a way that lowers the amount of memory needed to run them.

❖ 2.1.1. Types of Subprograms

There are two types of subprograms: procedures and functions. Procedures are usually used to perform actions, while functions are used to compute and return values.

2.2. Procedures

Procedures are subprograms that perform an action. They do not return a value.

The basic syntax of a procedure is shown below:

```
CREATE [OR REPLACE] PROCEDURE procedure_name(
  /* parameters */
)
IS
  /* declarations */
BEGIN
  /* executable commands */
EXCEPTION
  /* error handling code */
END procedure_name;
```

Evaluation Copy

The `OR REPLACE` is optional. If you do not include it and there is already a subprogram with the same name, running the `CREATE PROCEDURE` statement will cause an error.

Including the name of the procedure after the `END` statement is not required, but it helps provide a visual cue indicating what is ending here. In other words, `END procedure_name;` is more helpful than just `END;`.

IS vs. AS

Sometimes you will see `AS` used in place of `IS`. When used to create PL/SQL objects (e.g., subprograms, packages, and cursors), the two are synonyms.

Notice that the structure is similar to that of a regular PL/SQL block, but there is no `DECLARE` part. Instead, any declarations should be made after the `IS` keyword.

EVALUATION COPY: Not to be used in class.

The EXCEPTION part of the block is optional.

The following example shows how to create a simple "Hello, world!" procedure. The procedure itself is followed by several ways of calling the procedure:

Demo 2.1: Subprograms/Demos/hello.sql

```
1.   CREATE OR REPLACE PROCEDURE hello
2.   IS
3.   BEGIN
4.     DBMS_OUTPUT.PUT_LINE('Hello, world!');
5.   END hello;
6.   /
7.   -- Execute procedure outside of block
8.   EXECUTE hello;
9.
10.  EXEC hello;
11.
12.  -- Execute procedure within block
13.  BEGIN
14.    hello;
15.  END;
16.  /
```

Evaluation Copy

Code Explanation

As the procedure does not take any parameters, you do not (in fact, **cannot**) include parentheses after the procedure name. Beginning the procedure definition as follows would cause an error:

```
CREATE OR REPLACE PROCEDURE hello()
IS...
```

To execute the procedure outside of a PL/SQL block, call it with the EXECUTE or EXEC commands, which are synonyms:

```
EXECUTE hello;

EXEC hello;
```

As no parameters are passed in, you do not need parentheses when executing the procedure.

To execute the procedure within a PL/SQL block, call it directly:

```
BEGIN
  hello;
END;
```

2.3. Variable Declarations

The following example adds a variable declaration to the procedure:

Demo 2.2: Subprograms/Demos/hello-with-variables.sql

```
1.   CREATE OR REPLACE PROCEDURE hello
2.   IS
3.     l_subject VARCHAR2(20) := 'PL/SQL';
4.   BEGIN
5.     DBMS_OUTPUT.PUT_LINE('Hello, world! You are learning ' ||
6.       l_subject || ' really fast. Great job!');
7.   END hello;
8.
9.   -- Execute procedure outside of block
10.  EXECUTE hello;
11.
12.  -- Execute procedure within block
13.  BEGIN
14.    hello;
15.  END;
```

Evaluation Copy

Code Explanation

There is no `DECLARE` keyword used in creating subprograms. Declarations come between the `IS` and `BEGIN` keywords.

2.4. Parameters

The following example adds formal parameters to the procedure:

EVALUATION COPY: Not to be used in class.

Demo 2.3: Subprograms/Demos/hello-with-params.sql

```
1.   CREATE OR REPLACE PROCEDURE hello(
2.     greeting VARCHAR2,
3.     person   VARCHAR2
4.   )
5.   IS
6.     subject VARCHAR2(20) := 'PL/SQL';
7.   BEGIN
8.     DBMS_OUTPUT.PUT_LINE(greeting || ', ' || person ||
9.       '! You are learning ' || subject ||
10.      ' really fast. Great job!');
11.  END hello;
12.
13.  -- Execute procedure outside of block
14.  EXECUTE hello('Hello', 'Dolly');
15.
16.  -- Execute procedure within block
17.  BEGIN
18.    hello('Hello', 'Dolly');
19.  END;
```

Evaluation Copy

Code Explanation

Notice the data type is not *constrained* in the parameter declaration. You cannot do the following:

```
CREATE OR REPLACE PROCEDURE hello(
  greeting VARCHAR2(20),
  person   VARCHAR2(20)
)
```

❖ 2.4.1. Formal Parameters vs. Actual Parameters

There are two distinct but related types of parameters: *formal* and *actual*:

- *Formal parameters* are defined in the subprogram definition. Formal parameters are sometimes just called *parameters*.
- *Actual parameters* are the values passed into the subprogram and assigned to the formal parameters. Actual parameters are sometimes called *arguments*.

In the code sample above, the formal parameters are `greeting` and `person` and the actual parameters are `'Hello'` and `'Dolly'`.

2.5. Parameters with Default Values

Sometimes you will want your formal parameters to have default values. For example, take a look at the following code:

Demo 2.4: Subprograms/Demos/parameters-with-default-values.sql

```
1.   -- Create procedure
2.   CREATE OR REPLACE PROCEDURE hello(
3.     greeting VARCHAR2,
4.     person   VARCHAR2,
5.     subject  VARCHAR2 := 'SQL'
6.   )
7.   IS
8.   BEGIN
9.     DBMS_OUTPUT.PUT_LINE(greeting || ', ' || person ||
10.      '! You are learning ' || subject ||
11.      ' really fast. Great job!');
12.  END hello;
13.
14.  -- Call procedure
15.  BEGIN
16.    -- No value passed for subject
17.    hello('Hello', 'Dolly');
18.
19.    -- Value passed for subject
20.    hello('Hello', 'Dolly', 'PL/SQL');
21.  END;
```

Evaluation Copy

Code Explanation

When you run the block beneath the procedure definition, you'll get the following:

```
Hello, Dolly! You are learning SQL really fast. Great job!
Hello, Dolly! You are learning PL/SQL really fast. Great job!
```

1. The first call does not pass in a value for `subject`, and so the default 'SQL' is used.

EVALUATION COPY: Not to be used in class.

2. The second call passes in 'PL/SQL' for `subject`, which replaces the default value.

List Required Parameters First

Although it is not required, we highly recommend that you list all your *required* parameters (those without defaults) before the *optional* parameters (those with defaults). If you do not, you will not be able to take advantage of the defaults unless you use named notation, which we will cover shortly.

2.6. Parameter Modes

There are three parameter modes for subprograms:

1. `IN` - **Read-only**. The default mode. Used for passing a value to the subprogram for use within the subprogram.
2. `OUT` - **Write-only**. Used for passing a variable *with no value* to the subprogram, so that the subprogram can assign a value to the variable for use after the subprogram is executed. Any value the variable has at the time it is passed to the subprogram will be removed, and the variable value will be `NULL` until it is explicitly assigned a value in the subprogram.
3. `IN OUT` - **Read / Write**. Used for passing a variable *with a value* to the subprogram, so that the subprogram can both use the incoming value and assign a new value to the variable for use after the subprogram is executed. The value of the variable at the time it is passed to the subprogram will not be changed unless the subprogram explicitly changes it.

Evaluation Copy

Note that only `IN` parameters can have default values.

2.7. IN Mode

Our previous examples of subprograms with parameters used `IN` mode as we didn't specify a mode and `IN` is the default. Here is a procedure with `IN` explicitly specified:

Demo 2.5: Subprograms/Demos/hello-in-mode.sql

```
1.   -- Create procedure
2.   CREATE OR REPLACE PROCEDURE hello(
3.     greeting IN VARCHAR2,
4.     person   IN VARCHAR2,
5.     subject  IN VARCHAR2 := 'SQL'
6.   )
7.   IS
8.   BEGIN
9.     DBMS_OUTPUT.PUT_LINE(greeting || ', ' || person ||
10.      '! You are learning ' || subject ||
11.      ' really fast. Great job!');
12.  END hello;
13.
14.  -- Call procedure
15.  BEGIN
16.  -- No value passed for subject
17.  hello('Hello', 'Dolly');
18.
19.  -- Value passed for subject
20.  hello('Hello', 'Dolly', 'PL/SQL');
21.  END;
```

Evaluation Copy

Code Explanation

Again, this doesn't change how the subprogram works as `IN` is the default mode.

Let's take a look at another example:

EVALUATION COPY: Not to be used in class.

Demo 2.6: Subprograms/Demos/parameter-modes-in.sql

```
1.   -- Create procedure
2.   CREATE OR REPLACE PROCEDURE test_in(
3.     foo IN NUMBER
4.   )
5.   IS
6.   BEGIN
7.     DBMS_OUTPUT.PUT_LINE('Within procedure: ' || foo);
8.   END test_in;
9.
10.  -- Call procedure
11.  DECLARE
12.    l_bar NUMBER := 1;
13.  BEGIN
14.    DBMS_OUTPUT.PUT_LINE('Before procedure call: ' || TO_CHAR(l_bar));
15.    test_in(l_bar);
16.    DBMS_OUTPUT.PUT_LINE('After procedure call: ' || TO_CHAR(l_bar));
17.  END;
```

Code Explanation

Evaluation Copy

Things to notice:

1. In the calling block, the `l_bar` variable is assigned a value of `1` before it is passed to the `test_in()` subprogram.
2. We pass `l_bar` to `test_in()`. It will be assigned to the `foo` parameter. Because the parameter mode of `foo` is `IN`, we are not allowed to change the value of `foo` within the subprogram. The value of `l_bar` is `1` before the call to the procedure, within the procedure, and after the call to the procedure:

```
Before procedure call: 1
Within procedure: 1
After procedure call: 1
```

Important takeaway: When the mode of the parameter is `IN`, you cannot change the value of the parameter within the subprogram.

2.8. OUT Mode

Review the following code. It creates `test_out()`, which is the same as `test_in()` with one exception: the parameter mode of `foo` is `OUT`.

Demo 2.7: Subprograms/Demos/parameter-modes-out.sql

```
1.   -- Create procedure
2.   CREATE OR REPLACE PROCEDURE test_out(
3.     foo OUT NUMBER
4.   )
5.   IS
6.   BEGIN
7.     DBMS_OUTPUT.PUT_LINE('Within procedure: ' || foo);
8.   END test_out;
9.
10.  -- Call procedure
11.  DECLARE
12.    l_bar NUMBER := 1;
13.  BEGIN
14.    DBMS_OUTPUT.PUT_LINE('Before procedure call: ' || TO_CHAR(l_bar));
15.    test_out(l_bar);
16.    DBMS_OUTPUT.PUT_LINE('After procedure call: ' || TO_CHAR(l_bar));
17.  END;
```

Evaluation Copy

Code Explanation

Things to notice:

1. In the calling block, the `l_bar` variable is assigned a value of `1` before it is passed to the `test_out()` subprogram. When we output the value of `l_bar` before invoking the subprogram, we see that it contains `1`:

   ```
   Before procedure call: 1
   ```

2. We pass `l_bar` to `test_out()`. It will be assigned to the `foo` parameter. This time, because the parameter mode of `foo` is `OUT`, we **can** change the value of `foo` within the subprogram. However, we do not change it. But, even though we assigned the variable a value of `1` before passing it to the procedure, when we try to output its value within the procedure, nothing

EVALUATION COPY: Not to be used in class.

gets output. That's because the value of all parameters of mode `OUT` is `NULL`, whether or not the passed-in variable had a value before it was passed in:

```
Within procedure:
```

3. Likewise, when we try to output the value of `l_bar` after the call to `test_out()`, nothing gets output:

```
After procedure call:
```

That's because the value of `l_bar` changes based on what happens within the `test_out()` procedure. As soon as the procedure is called, it becomes `NULL`. And as nothing is assigned to `foo` within the `test_out()` subprogram, the value of `l_bar` remains `NULL`.

The following demo shows how you can change the value of the parameter within the subprogram:

Demo 2.8: Subprograms/Demos/parameter-modes-out-2.sql

Evaluation Copy

```
1.  -- Create procedure
2.  CREATE OR REPLACE PROCEDURE test_out(
3.    foo OUT NUMBER
4.  )
5.  IS
6.  BEGIN
7.    foo := 2;
8.    DBMS_OUTPUT.PUT_LINE('Within procedure: ' || foo);
9.    foo := 3;
10. END test_out;
11.
12. -- Call procedure
13. DECLARE
14.   l_bar NUMBER := 1;
15. BEGIN
16.   DBMS_OUTPUT.PUT_LINE('Before procedure call: ' || TO_CHAR(l_bar));
17.   test_out(l_bar);
18.   DBMS_OUTPUT.PUT_LINE('After procedure call: ' || TO_CHAR(l_bar));
19. END;
```

Code Explanation

Notice the value of `l_bar` is affected by changes to `foo`:

```
Before procedure call: 1
Within procedure: 2
After procedure call: 3
```

Important takeaway: When the mode of the parameter is `OUT`, any changes to the parameter within the subprogram will affect the passed-in variable. The subprogram immediately changes the value of the parameter to `NULL`. You can then make further changes, which will affect both the subprogram parameter and the passed-in variable.

2.9. IN OUT Mode

The `IN OUT` mode combines `IN` and `OUT`. The initial value of the parameter will be the same as the value of the passed-in variable. Here is the same subprogram using an `IN OUT` parameter:

Demo 2.9: Subprograms/Demos/parameter-modes-in-out.sql

Evaluation Copy

```
1.   -- Create procedure
2.   CREATE OR REPLACE PROCEDURE test_in_out(
3.     foo IN OUT NUMBER
4.   )
5.   IS
6.   BEGIN
7.     DBMS_OUTPUT.PUT_LINE('Within procedure: ' || foo);
8.     foo := 2;
9.   END test_in_out;
10.
11.  -- Call procedure
12.  DECLARE
13.    l_bar NUMBER := 1;
14.  BEGIN
15.    DBMS_OUTPUT.PUT_LINE('Before procedure call: ' || TO_CHAR(l_bar));
16.    test_in_out(l_bar);
17.    DBMS_OUTPUT.PUT_LINE('After procedure call: ' || TO_CHAR(l_bar));
18.  END;
```

Code Explanation

This time, we output the value of `foo` before changing it within the subprogram. Notice that it retains the value of the passed-in `l_bar` variable:

EVALUATION COPY: Not to be used in class.

```
Before procedure call: 1
Within procedure: 1
```

We then assign `2` to `foo` after outputting it within the subprogram. That change affects the value of `l_bar`:

```
After procedure call: 2
```

Important takeaway: When the mode of the parameter is `IN OUT`, the subprogram retains the value of the passed-in variable, but you can change that value, which will affect both the subprogram parameter and the passed-in variable.

IN Mode
With the `IN` mode, you don't have to pass in a variable. You can pass in a literal value. However, with the `OUT` and `IN OUT` modes, you have to pass in a variable to store the `OUT` value.

Evaluation Copy

2.10. Named Notation

All of the examples in this section are in the `Subprograms/Demos/named-notation.sql` file. Please open that file in SQL Developer so you can run the code shown to see the results.

Up until this point, we have been using *positional notation* when calling subprograms. This means that we have been matching the position of the actual parameters in the call to that of the formal parameters in the subprogram definition. For example, we would call a procedure that starts as follows:

```
CREATE OR REPLACE PROCEDURE hello(
  greeting IN VARCHAR2,
  person   IN VARCHAR2,
  subject  IN VARCHAR2 := 'SQL'
)
```

...like this...

```
hello('Howdy', 'Dolly', 'PL/SQL');
```

Where `greeting` would get 'Howdy', `person` would get 'Dolly', and `subject` would get 'PL/SQL', all based on the position of the values in the call. While using positional notation works fine, it has the disadvantage of being a little opaque. Unless you are familiar with the design of the subprogram, you cannot tell whether you will be saying "Howdy" to Dolly or "Dolly" to Howdy.

Using *named notation* is clearer:

```
hello(greeting => 'Howdy',
            person   => 'Dolly',
            subject  =>'PL/SQL');
```

The `=>` sign is used to pair a formal parameter with an actual parameter.

With named notation, you can change the position of the parameters:

Evaluation Copy

```
hello(subject  => 'PL/SQL',
            greeting => 'Howdy',
            person   => 'Dolly');
```

❖ 2.10.1. Mixed Notation

You can combine named notation and positional notation, resulting it what is known as `mixed notation`:

```
hello('Howdy',
            'Dolly',
            subject  => 'PL/SQL');
```

Again, we encourage you to open `Subprograms/Demos/named-notation.sql` to run the examples we have just shown.

EVALUATION COPY: Not to be used in class.

2.11. Using SQL in a Subprogram

All of the examples in this section are in the `Subprograms/Demos/embedded-sql.sql` file. Please open that file in SQL Developer so you can run the code shown to see the results.

Consider the following procedure:

```
CREATE OR REPLACE PROCEDURE assign_country_name(
  country_id     IN      CHAR,
  country_name   OUT     VARCHAR2
) IS
  -- No variable declarations
BEGIN
  SELECT country_name
  INTO country_name
  FROM countries
  WHERE country_id = country_id;
END assign_country_name;
```

This procedure could be called like this:

Evaluation Copy

```
assign_country_name(
  country_id    => country_id,
  country_name  => country_name
);
```

The purpose of the `assign_country_name()` procedure is to assign a value to the passed-in `country_name` variable based on the passed-in `country_id`. But there is a problem with this line of code:

```
WHERE country_id = country_id;
```

What does the second `country_id` reference? Our intention is for it to reference the passed-in `country_id` parameter, but the `countries` table that we're querying has a column named `country_id` and it will first look there before looking for a variable outside of the SQL query. As such, the `WHERE` clause is returning rows in which the `country_id` field equals itself, which is true for all rows. So, it will return every row in the `countries` table. Give it a try. Run the following query:

```
SELECT country_name
FROM countries
WHERE country_id = country_id;
```

The result will be (first 10 rows shown):

	COUNTRY_NAME
1	Argentina
2	Australia
3	Belgium
4	Brazil
5	Canada
6	Switzerland
7	China
8	Germany
9	Denmark
10	Egypt

Evaluation Copy

As the `SELECT...INTO` statement in the `assign_country_name()` procedure expects only one row, this will cause an error.

There are a couple of ways to fix this:

1. Qualify the variable name with the name of the subprogram using dot notation (i.e., separating the qualifier and the variable with a dot):

```
CREATE OR REPLACE PROCEDURE assign_country_name(
  country_id     IN      CHAR,
  country_name   OUT     VARCHAR2
) IS
-- No variable declarations
BEGIN
  SELECT country_name
  INTO assign_country_name.country_name
  FROM countries
  WHERE country_id = assign_country_name.country_id;
END assign_country_name;
```

   A. By qualifying `country_id` with `assign_country_name` in `WHERE country_id = assign_country_name.country_id`, we make it clear that we want the

EVALUATION COPY: Not to be used in class.

`country_id` that is a parameter or variable from the `assign_country_name()` procedure, and not the field in the `countries` table.

B. Although not strictly required, we also qualified `country_name` in `INTO assign_country_name.country_name` to make it clear that we are selecting the value into the procedure's `country_name` parameter or variable.

2. Remove the ambiguity by giving the parameter a different name:

```
CREATE OR REPLACE PROCEDURE assign_country_name(
  country_id_in     IN      CHAR,
  country_name_out  OUT     VARCHAR2
) IS
  -- No variable declarations
BEGIN
  SELECT country_name
  INTO country_name_out
  FROM countries
  WHERE country_id = country_id_in;
END assign_country_name;
```

This is our preferred method for making sure that the incoming parameters have distinct names. We usually use suffixes that identify the type of parameter:

Evaluation Copy

- `_in` for **IN** parameters. For example: `country_id_in`, `country_name_in`.
- `_out` for **OUT** parameters. For example: `country_id_out`, `country_name_out`.
- `_io` for **IN OUT** parameters. For example: `country_id_io`, `country_name_io`.

2.12. %TYPE

When a parameter or variable is going to be used for a value of a field in a table, you can reference that field to make sure the data type of the parameter matches the data type of the field. You do that using the following syntax:

```
table_name.column_name%TYPE
```

Consider the following example:

Demo 2.10: Subprograms/Demos/assign-country-name-type.sql

```
1.   CREATE OR REPLACE PROCEDURE assign_country_name(
2.     country_id_in     IN      countries.country_id%TYPE,
3.     country_name_out  OUT     countries.country_name%TYPE
4.   ) IS
5.     -- No variable declarations
6.   BEGIN
7.     SELECT country_name
8.     INTO country_name_out
9.     FROM countries
10.    WHERE country_id = country_id_in;
11.  END assign_country_name;
```

Evaluation Copy

Code Explanation

Instead of explicitly assigning data types to `country_id_in` and `country_name_out`, we use the `%TYPE` attribute to match the data types of the columns. This way, if the column data types are altered, we won't have to update the procedure.

Later, we will meet the `%ROWTYPE` attribute, which specifies that a variable represents a record in a table, view or cursor.

EVALUATION COPY: Not to be used in class.

Exercise 2: Creating a Procedure

20 to 30 minutes

In this exercise you will create a procedure that updates an employee's email address.

1. Open `Subprograms/Exercises/update-employee-email.sql` in SQL Developer.
2. Notice that it already has code that calls a subprogram called `update_employee_email()`, passing in four parameters. After calling it, it outputs something like:

```
Email of Peter Tucker changed from "PTUCKER" to "peter.tucker@hr".
```

3. Your job is to create the `update_employee_email()` procedure, which should take four parameters:
    A. `employee_id_in` - for receiving the `employee_id` of the employee whose email is being changed.
    B. `new_email_io` - for receiving the `email` of the employee whose email is being changed. This should be writable as well, so that we can confirm that the change has taken effect.
    C. `old_email_out` - for writing the employee's old email to a passed-in variable.
    D. `full_name_out` - for writing the employee's full name to a passed-in variable.
4. In the main body of the procedure, you should:
    A. Select the employee's current email into the `old_email_out` parameter.
    B. Update the employee's email to the `new_email_io` passed in concatenated with '@hr', so that 'peter.tucker' becomes 'peter.tucker@hr'. **Be careful to only update the employee who matches the passed-in id.**
    C. Select the employee's first and last name (concatenated together with a space in between) into `full_name_out` and the employee's updated email into `new_email_io`.
5. Run your procedure to create it and then run the block below it. It should output the following without any errors:

```
Email of Peter Tucker changed from "PTUCKER" to "peter.tucker@hr".
```

6. If you get any errors, fix your subprogram and try again.

Evaluation Copy

Solution: Subprograms/Solutions/update-employee-email.sql

```
1.    -- Create the procedure here
2.    CREATE OR REPLACE PROCEDURE update_employee_email (
3.      employee_id_in  IN      employees.employee_id%TYPE,
4.      new_email_io    IN OUT  employees.email%TYPE,
5.      old_email_out   OUT     employees.email%TYPE,
6.      full_name_out   OUT     VARCHAR2
7.    ) IS
8.      -- No variable declarations
9.    BEGIN
10.     SELECT email
11.     INTO old_email_out
12.     FROM employees
13.     WHERE employee_id = employee_id_in;
14.
15.     UPDATE employees
16.     SET email = new_email_io || '@hr'
17.     WHERE employee_id = employee_id_in;
18.
19.     SELECT first_name || ' ' || last_name, email
20.     INTO
21.       full_name_out,
22.       new_email_io
23.     FROM employees
24.     WHERE employee_id = employee_id_in;
25.   END update_employee_email;
      -------Lines 26 through 42 Omitted-------
```

Evaluation Copy

Code Explanation

Note that assigning a value to a field (e.g., 'peter.tucker@hr') that is different from the one the user thinks they assigned (e.g., 'peter.tucker') is not good practice. We do it here only to illustrate that you can change the value of an `IN OUT` parameter.

2.13. Functions

Functions are coded similarly to procedures, but functions must include a `RETURN` clause as functions return values.

The basic syntax of a function is shown below:

EVALUATION COPY: Not to be used in class.

```
CREATE [OR REPLACE] FUNCTION function_name(
  /* parameters */
)
  RETURN data_type
IS
  /* declarations */
BEGIN
  /* executable commands */
  /* must return value */
EXCEPTION
  /* error handling code */
END function_name;
```

Evaluation Copy

The code below shows two simple functions. The first returns the circumference of a circle. The second returns the name of a country based on the `country_id`.

Demo 2.11: Subprograms/Demos/functions.sql

```
1.  CREATE OR REPLACE FUNCTION get_circumference(
2.    radius_in IN NUMBER
3.  )
4.    RETURN FLOAT
5.  IS
6.    c_pi CONSTANT FLOAT := 3.14159;
7.  BEGIN
8.    RETURN c_pi * (radius_in * 2);
9.  END get_circumference;
10. /
11. CREATE OR REPLACE FUNCTION get_country_name(
12.   country_id_in IN CHAR
13. )
14.   RETURN VARCHAR2
15. IS
16.   l_country_name VARCHAR2(40);
17. BEGIN
18.   SELECT country_name
19.   INTO l_country_name
20.   FROM countries
21.   WHERE country_id = country_id_in;
22.
23.   RETURN l_country_name;
24. END get_country_name;
25. /
26. DECLARE
27.   l_radius        NUMBER  := 5;
28.   l_circumference FLOAT;
29.   l_country_id    CHAR(2) := 'US';
30.   l_country_name  VARCHAR2(40);
31. BEGIN
32.   l_circumference := get_circumference(l_radius);
33.   l_country_name  := get_country_name(l_country_id);
34.
35.   DBMS_OUTPUT.PUT_LINE('Circumference: ' || l_circumference);
36.   DBMS_OUTPUT.PUT_LINE('Hello, ' || l_country_name || '!');
37. END;
38. /
```

Evaluation Copy

Code Explanation

After creating the two functions, we call them both in a block and output the returned values. It will output:

EVALUATION COPY: Not to be used in class.

```
Circumference: 31.4159
Hello, United States of America!
```

Evaluation Copy

Exercise 3: Creating a Function

20 to 30 minutes

In this exercise you will create a function that returns the full name of an employee's manager.

1. Open `Subprograms/Exercises/get-manager.sql` in SQL Developer.
2. Notice that it already has code that calls a function called `get_manager()`, passing in two parameters, and assigning the results to a variable. After calling it, it outputs something like:

```
Neena Kochhar is the manager of Nancy Greenberg.
```

Evaluation Copy

3. Your job is to create the `get_manager()` function. If you need help with the SQL query to get an employee's manager, open `Subprograms/Exercises/select-manager-query.sql` to use as a guide.
4. Run your function code to create it and then run the block below it. It should output the following without any errors:

```
Neena Kochhar is the manager of Nancy Greenberg.
```

5. If you get any errors, fix your function and try again.

EVALUATION COPY: Not to be used in class.

Solution: Subprograms/Solutions/get-manager.sql

```
1.    -- Create the function here
2.    CREATE OR REPLACE FUNCTION get_manager(
3.      first_name_in  IN  employees.first_name%TYPE,
4.      last_name_in   IN  employees.last_name%TYPE
5.    )
6.     RETURN VARCHAR2
7.    IS
8.      l_manager_name VARCHAR2(50);
9.    BEGIN
10.     SELECT first_name || ' ' || last_name
11.     INTO l_manager_name
12.     FROM employees
13.     WHERE employee_id = (
14.       SELECT manager_id
15.       FROM employees
16.       WHERE first_name = first_name_in
17.         AND last_name = last_name_in
18.     );
19.
20.     RETURN l_manager_name;
21.   END get_manager;
      -------Lines 22 through 35 Omitted-------
```

Evaluation Copy

Code Explanation

Note that this function will error if you pass in a name that doesn't match an employee in the `employees` table or matches an employee who does not have a manager. To test this, pass in your own name to the function. You will get an error like this:

EVALUATION COPY: Not to be used in class.

```
Error starting at line : 22 in command -
DECLARE
  l_first_name    VARCHAR2(20);
  l_last_name     VARCHAR2(25);
  l_manager_name  VARCHAR2(45);
BEGIN
  l_first_name    := 'Nat';
  l_last_name     := 'Dunn';
  l_manager_name  := get_manager(first_name, last_name);

  DBMS_OUTPUT.PUT_LINE(l_manager_name || ' is the manager of ' ||
    l_first_name || ' ' || l_last_name || '.');
END;
Error report -
ORA-01403: no data found
ORA-06512: at "HR.GET_MANAGER", line 9
ORA-06512: at line 8
01403. 00000 -  "no data found"
*Cause:    No data was found from the objects.
*Action:   There was no data from the objects which may be due to end of fetch.
```

Evaluation Copy

The function will also error if there are two employees with the same first and last names.

We will learn how to deal with errors like these in upcoming lessons.

2.14. Using PL/SQL Functions in SQL Queries

You can use PL/SQL functions in SQL queries as long as those functions return SQL data types. A function that returns a `BOOLEAN` cannot be used, because SQL doesn't support `BOOLEAN`.

To see how this works, try running the following demo:

Demo 2.12: Subprograms/Demos/get-manager-in-sql-query.sql

```
1.   SELECT first_name, last_name,
2.     get_manager(first_name, last_name) AS manager
3.   FROM employees;
```

Code Explanation

The first 10 rows of the results are shown below:

	FIRST_NAME	LAST_NAME	MANAGER
1	Steven	King	(null)
2	Neena	Kochhar	Steven King
3	Lex	De Haan	Steven King
4	Alexander	Hunold	Lex De Haan
5	Bruce	Ernst	Alexander Hunold
6	David	Austin	Alexander Hunold
7	Valli	Pataballa	Alexander Hunold
8	Diana	Lorentz	Alexander Hunold
9	Nancy	Greenberg	Neena Kochhar
10	Daniel	Faviet	Nancy Greenberg

Avoid OUT Parameters with Functions

Although it is not technically wrong, you should not use `OUT` or `IN OUT` parameters with functions. Functions are meant to receive zero or more parameters and return a single value.

Evaluation Copy

2.15. Dropping a Subprogram

Dropping (deleting) a procedure or function is easy. It is just:

```
DROP PROCEDURE procedure_name;
```

```
DROP FUNCTION function_name;
```

For example:

```
DROP PROCEDURE hello;
```

Executing this will result in the following output:

EVALUATION COPY: Not to be used in class.

```
Procedure HELLO dropped.
```

Conclusion

Evaluation Copy

In this lesson, you have learned to create the two types of subprograms: procedures and functions.

EVALUATION COPY: Not to be used in class.

LESSON 3
Conditional Processing

Topics Covered

☑ `BOOLEAN` values and expressions.

☑ `IF` conditions.

☑ `CASE` statements.

Introduction

In this lesson, you will learn about `BOOLEAN` values and expressions and how to write `IF` conditions and `CASE` statements.

Evaluation Copy

3.1. Conditions and Booleans

In programming, a condition is a statement that evaluates to `TRUE` or `FALSE`. This might be a comparison between two values (e.g., `a = b` or `a < b`) or it might be the result of a function call (e.g., `is_manager()`). An expression that evaluates to `TRUE` or `FALSE` is called a *Boolean* expression. In addition, PL/SQL includes a `BOOLEAN` data type that is not part of standard SQL.

There are two main types of conditionals in PL/SQL:

1. `IF-ELSIF-ELSE`
2. `CASE-WHEN-ELSE`

3.2. IF-ELSIF-ELSE Conditions

The syntax for an `IF-ELSIF-ELSE` condition is as follows:

```
IF condition THEN
  /* executable commands */
ELSIF condition THEN
  /* executable commands */
ELSIF condition THEN
  /* executable commands */
ELSE
  /* executable commands */
END IF;
```

`ELSIF` is short for **ELSE IF**. There can be zero or more `ELSIF` blocks. The `ELSE` block is optional.

The table below shows the comparison operators used in conditional statements:

Comparison Operators

Operator	Description
`IS NULL`	`NULL` check. Returns `TRUE` if the value is `NULL` and `FALSE` if it isn't.
`IS NOT NULL`	`NOT NULL` check. Returns `FALSE` if the value is `NULL` and `TRUE` if it isn't.
`=`	Equals
`<>`, `!=`, `~=`, `^=`	Doesn't equal
`>`	Is greater than
`<`	Is less than
`>=`	Is greater than or equal to
`<=`	Is less than or equal to
`LIKE`, `BETWEEN`, and `IN`	Work exactly as they do in standard SQL.

Evaluation Copy

The following code shows how to use simple `IF - ELSE` statements:

EVALUATION COPY: Not to be used in class.

Demo 3.1: Conditional-Processing/Demos/if-else.sql

```
1.   CREATE OR REPLACE FUNCTION can_buy_alcohol(
2.     age_in IN  NUMBER
3.   )
4.     RETURN BOOLEAN
5.   IS
6.   BEGIN
7.     IF age_in >= 21 THEN
8.       RETURN TRUE;
9.     ELSE
10.      RETURN FALSE;
11.    END IF;
12.  END can_buy_alcohol;
13.
14.  DECLARE
15.    l_age NUMBER(6,0);
16.  BEGIN
17.    l_age := 22;
18.    IF can_buy_alcohol(l_age) THEN
19.      DBMS_OUTPUT.PUT_LINE('You can buy alcohol.');
20.    ELSE
21.      DBMS_OUTPUT.PUT_LINE('You cannot buy alcohol.');
22.    END IF;
23.  END;
```

Evaluation Copy

Code Explanation

Notice that there are two `IF - ELSE` statements here. One in the `can_buy_alcohol()` function and the other in the block below it.

As we mentioned earlier, a condition is a statement that evaluates to `TRUE` or `FALSE`. Consider the `age_in >= 21` condition in the `can_buy_alcohol()` function. We use an `IF` condition to check if that condition is `TRUE` or `FALSE` and then, based on the result, return `TRUE` or `FALSE`. Let's break that down:

```
IF age_in >= 21 THEN
  RETURN TRUE;
ELSE
  RETURN FALSE;
END IF;
```

If `age_in` is greater than or equal to `21`, then `age_in >= 21` evaluates to `TRUE`, which can be expressed like this:

```
IF TRUE THEN
  RETURN TRUE;
ELSE
  RETURN FALSE;
END IF;
```

Can you see how that's a little silly? We don't need an `IF` condition here at all. We could simply rewrite this as:

Evaluation Copy

```
RETURN TRUE;
```

Backing that up one step, we know that `age_in >= 21` will evaluate to either `TRUE` or `FALSE`, which are the two possible values that can be returned by the `can_buy_alcohol()` function. So, instead of using an `IF` condition, we can just return the conditional expression:

```
RETURN age_in >= 21;
```

The code below shows the full function rewritten:

EVALUATION COPY: Not to be used in class.

Demo 3.2: Conditional-Processing/Demos/return-conditional-expression.sql

```
1.   CREATE OR REPLACE FUNCTION can_buy_alcohol(
2.     age_in IN NUMBER
3.   )
4.     RETURN BOOLEAN
5.   IS
6.   BEGIN
7.     RETURN age_in >= 21;
8.   END can_buy_alcohol;
9.
10.  DECLARE
11.    l_age NUMBER(6,0);
12.  BEGIN
13.    l_age := 22;
14.    IF can_buy_alcohol(l_age) THEN
15.      DBMS_OUTPUT.PUT_LINE('You can buy alcohol.');
16.    ELSE
17.      DBMS_OUTPUT.PUT_LINE('You cannot buy alcohol.');
18.    END IF;
19.  END;
```

Evaluation Copy

Exercise 4: Creating a get_age() Function

15 to 25 minutes

In this exercise you will create a `get_age()` function that takes one parameter: a birth date, and returns the age of the person born that day in years as a `FLOAT`.

There are different ways to calculate the number of years between two dates:

1. Subtracting one date from another gives you the number of days between the two dates. You can then divide that number by `365.25` to get the number of years between the two dates.
2. The built-in `MONTHS_BETWEEN()` function takes two dates and returns the number of months between the two. You can divide that result by `12` to get the number of years between the two dates.

Evaluation Copy

Open `Conditional-Processing/Exercises/get-age.sql` in SQL Developer.

1. Create a function called `get_age()` that takes one parameter: a birth date, and returns the age of the person born that day in years as a `FLOAT`.
2. In the block below your function, write code to output 'You can buy alcohol.' or 'You cannot buy alcohol.' based on the value of the `l_birth_date` variable. Make sure to use your new `get_age()` function you wrote and the `can_buy_alcohol()` function that we created earlier in this lesson.

EVALUATION COPY: Not to be used in class.

Solution: Conditional-Processing/Solutions/get-age.sql

```
1.  CREATE OR REPLACE FUNCTION get_age(
2.    birth_date_in IN  DATE
3.  )
4.    RETURN FLOAT
5.  IS
6.    l_age FLOAT;
7.  BEGIN
8.    l_age := MONTHS_BETWEEN(CURRENT_DATE, birth_date_in)/12;
9.    -- Uncomment the line below to output age
10.   -- DBMS_OUTPUT.PUT_LINE('Your age: ' || l_age);
11.   RETURN l_age;
12. END get_age;
13.
14. DECLARE
15.   l_birth_date  DATE;
16.   l_age         FLOAT;
17. BEGIN
18.   l_birth_date := TO_DATE('1986-06-03', 'YYYY-MM-DD');
19.   l_age        := get_age(l_birth_date);
20.   IF can_buy_alcohol(l_age) THEN
21.     DBMS_OUTPUT.PUT_LINE('You can buy alcohol.');
22.   ELSE
23.     DBMS_OUTPUT.PUT_LINE('You cannot buy alcohol.');
24.   END IF;
25. END;
```

3.3. ELSIF

An `IF` statement can take zero or more `ELSIF` conditions. The following demo, which shows how to create a `get_quarter()` function, illustrates this:

EVALUATION COPY: Not to be used in class.

Demo 3.3: Conditional-Processing/Demos/get-quarter.sql

```
1.   CREATE OR REPLACE FUNCTION get_quarter(
2.     date_in IN DATE
3.   )
4.     RETURN NUMBER
5.   IS
6.     l_m CHAR(3);
7.   BEGIN
8.     l_m := TO_CHAR(date_in, 'Mon');
9.     IF l_m IN ('Jan','Feb','Mar') THEN
10.      RETURN 1;
11.    ELSIF l_m IN ('Apr','May','Jun') THEN
12.      RETURN 2;
13.    ELSIF l_m IN ('Jul','Aug','Sep') THEN
14.      RETURN 3;
15.    ELSE
16.      RETURN 4;
17.    END IF;
18.  END get_quarter;
19.
20.  DECLARE
21.    l_the_date  DATE;
22.    l_quarter   NUMBER(1,0);
23.  BEGIN
24.    l_the_date  := CURRENT_DATE;
25.    l_quarter   := get_quarter(l_the_date);
26.    DBMS_OUTPUT.PUT_LINE(l_quarter);
27.  END;
```

Evaluation Copy

Exercise 5: Creating a check_rights() Procedure

20 to 30 minutes

In this exercise you will create a `check_rights()` procedure that takes two parameters: a `DATE` indicating a person's birth date and a `BOOLEAN` indicating whether or not the person has a driver's license.

The procedure will output one of the following messages:

1. You can drive and drink, but don't drink and drive. - For people over 21 who have a driver's license.
2. You can drink, but cannot drive. - For people over 21 who do not have a driver's license.
3. You can drive, but cannot drink. - For people between 16 and 21 who have a driver's license.
4. How did you get your license so young? - For people younger than 16 who have a driver's license.
5. You cannot drink or drive. - For everyone else. (People younger than 21 who do not have a driver's license.)

Evaluation Copy

Open `Conditional-Processing/Exercises/check-rights.sql` in SQL Developer.

1. Create a function called `check_rights()` that takes two parameters: a `DATE` indicating a person's birth date and a `BOOLEAN` indicating whether or not the person has a driver's license.
2. Use the `get_age()` function from the previous exercise to calculate the age based on the passed-in birth date.
3. Write an `IF` - `ELSIF` - `ELSE` condition to output the appropriate message based on the passed-in values.
4. Run the code to create the procedure and then run the block of code already in the file to test your solution.
5. If you get any errors, fix them and try again.

EVALUATION COPY: Not to be used in class.

Solution: Conditional-Processing/Solutions/check-rights.sql

```
1.  CREATE OR REPLACE PROCEDURE check_rights(
2.    birth_date_in  IN  DATE,
3.    has_license_in IN  BOOLEAN
4.  )
5.  IS
6.    l_age NUMBER(6,2);
7.  BEGIN
8.    l_age := get_age(birth_date_in);
9.    IF l_age >= 21 AND has_license_in THEN
10.     DBMS_OUTPUT.PUT_LINE('You can drive and drink, ' ||
11.       'but don''t drink and drive.');
12.   ELSIF l_age >= 21 THEN
13.     DBMS_OUTPUT.PUT_LINE('You can drink, but cannot drive.');
14.   ELSIF l_age >= 16 AND has_license_in THEN
15.     DBMS_OUTPUT.PUT_LINE('You can drive, but cannot drink.');
16.   ELSIF has_license_in THEN -- Must be younger than 16
17.     DBMS_OUTPUT.PUT_LINE('How did you get your license so young?');
18.   ELSE -- Younger than 16 and no license
19.     DBMS_OUTPUT.PUT_LINE('You cannot drink or drive.');
20.   END IF;
21. END check_rights;
22.
23. DECLARE
24.   l_birth_date DATE;
25.   l_has_license BOOLEAN;
26. BEGIN
27.   l_birth_date := TO_DATE('1986-06-03', 'YYYY-MM-DD');
28.   l_has_license := FALSE;
29.   check_rights(l_birth_date, l_has_license);
30. END;
```


EVALUATION COPY: Not to be used in class.

Exercise 6: Creating an is_manager() Function

25 to 40 minutes

Consider the two SQL queries shown below:

```
-- Query 1
SELECT COUNT(employee_id) AS num_reports
FROM employees
WHERE manager_id = 100;

-- Query 2
SELECT COUNT(employee_id) AS num_reports
FROM employees
WHERE manager_id = (
  SELECT employee_id
  FROM employees
  WHERE first_name = 'Steven'
    AND last_name = 'King'
);
```

Evaluation Copy

Both queries will return the number of people who report to Steven King, whose employee id is 100.

If `num_reports` is `0`, it would indicate that Steven King is not a manager. We could write a SQL query to return just that information:

```
SELECT CASE COUNT(employee_id)
  WHEN 0 THEN 'Not manager'
  ELSE 'Manager'
  END AS is_manager
FROM employees
WHERE manager_id = 100;
```

In this exercise, you will create a PL/SQL `is_manager()` function that returns `TRUE` if the employee has any direct reports and `FALSE` if not. The function should be callable in two ways:

```
-- Call 1
is_manager(
  employee_id_in => l_employee_id
);

-- Call 2
is_manager(
  first_name_in => l_first_name,
  last_name_in  => l_last_name
);
```

Evaluation Copy

The starting code, shown below, contains two PL/SQL blocks that make the two different calls. Your job is to write the function.

EVALUATION COPY: Not to be used in class.

Exercise Code 6.1: Conditional-Processing/Exercises/is-manager.sql

```
1.   -- Create the is_manager() function here
2.
3.   DECLARE
4.     l_employee_id NUMBER(6,0);
5.     l_is_mgr      BOOLEAN;
6.   BEGIN
7.     l_employee_id := 100;
8.     l_is_mgr := is_manager(
9.       employee_id_in => l_employee_id
10.    );
11.
12.    IF l_is_mgr THEN
13.      DBMS_OUTPUT.PUT_LINE('Is manager');
14.    ELSE
15.      DBMS_OUTPUT.PUT_LINE('Is not manager');
16.    END IF;
17.  END;
18.
19.  DECLARE
20.    l_first_name  VARCHAR2(20);
21.    l_last_name   VARCHAR2(25);
22.    l_is_mgr      BOOLEAN;
23.  BEGIN
24.    l_first_name :=  'Steven';
25.    l_last_name :=  'King';
26.    l_is_mgr := is_manager(
27.      first_name_in => l_first_name,
28.      last_name_in  => l_last_name
29.    );
30.
31.    IF l_is_mgr THEN
32.      DBMS_OUTPUT.PUT_LINE(l_first_name || ' ' || l_last_name ||
33.        ' is a manager');
34.    ELSE
35.      DBMS_OUTPUT.PUT_LINE(l_first_name || ' ' || l_last_name ||
36.        ' is NOT a manager');
37.    END IF;
38.  END;
```

Evaluation Copy

1. Open `Conditional-Processing/Exercises/is-manager.sql` in SQL Developer.
2. Write an `is_manager()` function that returns a `BOOLEAN`. The function should allow for three parameters, all of which should default to `NULL`:

A. `employee_id_in`

B. `first_name_in`

C. `last_name_in`

3. If a non-`NULL` `employee_id_in` is passed in, the function should use `employee_id_in` to identify the person and find out if they have direct reports. Otherwise, the function should use the person's first and last names.
4. When you're finished, test your function by running the two blocks of code below it. Try passing in some other data. You should find out that:

Evaluation Copy

A. The employee with the id of `104` is not a manager.

B. The employee with the id of `145` is a manager.

C. Diana Lorentz is not a manager.

D. Eleni Zlotkey is a manager.

EVALUATION COPY: Not to be used in class.

Solution: Conditional-Processing/Solutions/is-manager.sql

```
1.   CREATE OR REPLACE FUNCTION is_manager(
2.     employee_id_in IN employees.employee_id%TYPE  := NULL,
3.     first_name_in  IN employees.first_name%TYPE   := NULL,
4.     last_name_in   IN employees.last_name%TYPE    := NULL
5.   )
6.    RETURN BOOLEAN
7.   IS
8.     l_num_reports NUMBER;
9.   BEGIN
10.    IF employee_id_in IS NOT NULL THEN
11.      SELECT COUNT(employee_id)
12.      INTO l_num_reports
13.      FROM employees
14.      WHERE manager_id = employee_id_in;
15.
16.      RETURN l_num_reports > 0;
17.    ELSE
18.      SELECT COUNT(employee_id)
19.      INTO l_num_reports
20.      FROM employees
21.      WHERE manager_id = (
22.        SELECT employee_id
23.        FROM employees
24.        WHERE first_name = first_name_in
25.          AND last_name = last_name_in
26.      );
27.
28.      RETURN l_num_reports > 0;
29.    END IF;
30.  END is_manager;
     -------Lines 31 through 67 Omitted-------
```

Evaluation Copy

3.4. BOOLEAN Values and Standard SQL

As we have mentioned, standard SQL does not support `BOOLEAN` values. As our `is_manager()` function returns a `BOOLEAN` value, we cannot use it in a SQL query. The following, for example, would cause an "invalid datatype" error:

EVALUATION COPY: Not to be used in class.

```
SELECT first_name, last_name, is_manager(employee_id)
FROM employees;
```

One workaround would be to change the `is_manager()` function so that it returns a `NUMBER` (e.g., `1` for `TRUE` and `0` for `FALSE`) or a `VARCHAR2` (e.g., `'true'` for `TRUE` and `'false'` for `FALSE`). That might be okay, but it sort of kills the purpose of using `BOOLEAN` values in PL/SQL, which is to get a simple `TRUE`/`FALSE` value returned.

Another solution that works in this case is to create a similar function that does something slightly different, but provides the same and additional information: a `num_reports()` function, like the one below:

Demo 3.4: Conditional-Processing/Demos/num-reports.sql

```
1.   CREATE OR REPLACE FUNCTION num_reports(
2.     employee_id_in IN  employees.employee_id%TYPE  := NULL,
3.     first_name_in  IN  employees.first_name%TYPE   := NULL,
4.     last_name_in   IN  employees.last_name%TYPE    := NULL
5.   )
6.    RETURN NUMBER
7.   IS
8.     l_num_rpts NUMBER;
9.   BEGIN
10.    IF employee_id_in IS NOT NULL THEN
11.      SELECT COUNT(employee_id)
12.      INTO l_num_rpts
13.      FROM employees
14.      WHERE manager_id = employee_id_in;
15.    ELSE
16.      SELECT COUNT(employee_id)
17.      INTO l_num_rpts
18.      FROM employees
19.      WHERE manager_id = (
20.        SELECT employee_id
21.        FROM employees
22.        WHERE first_name = first_name_in
23.          AND last_name = last_name_in);
24.    END IF;
25.
26.    RETURN l_num_rpts;
27.  END num_reports;
```

Evaluation Copy

Code Explanation

Instead of returning `TRUE` or `FALSE`, this function returns the number of direct reports that this employee has.

Evaluation Copy

The `num_reports()` function can be called the same ways in which the `is_manager()` function is called. The return value will be `1` or higher for managers and `0` for non-managers:

EVALUATION COPY: Not to be used in class.

Demo 3.5: Conditional-Processing/Demos/num-reports-call.sql

```
1.   DECLARE
2.     l_employee_id NUMBER(6,0);
3.     l_num_rpts    NUMBER(6,0);
4.   BEGIN
5.     l_employee_id := 100;
6.     l_num_rpts := num_reports(
7.       employee_id_in => l_employee_id
8.     );
9.
10.    IF l_num_rpts >= 1 THEN
11.      DBMS_OUTPUT.PUT_LINE('A manager with ' ||
12.        l_num_rpts || ' direct reports.');
13.    ELSE
14.      DBMS_OUTPUT.PUT_LINE('Not a manager');
15.    END IF;
16.  END;
17.
18.  DECLARE
19.    l_first_name  VARCHAR2(20);
20.    l_last_name   VARCHAR2(25);
21.    l_num_rpts    NUMBER(6,0);
22.  BEGIN
23.    l_first_name :=  'Steven';
24.    l_last_name  :=  'King';
25.    l_num_rpts   := num_reports(
26.      first_name_in => l_first_name,
27.      last_name_in  => l_last_name
28.    );
29.
30.    IF l_num_rpts >= 1 THEN
31.      DBMS_OUTPUT.PUT_LINE(l_first_name || ' ' || l_last_name ||
32.        ' is a manager with ' || l_num_rpts || ' direct reports.');
33.    ELSE
34.      DBMS_OUTPUT.PUT_LINE(l_first_name || ' ' || l_last_name ||
35.        ' is NOT a manager');
36.    END IF;
37.  END;
```

Evaluation Copy

Code Explanation

The upper block will output:

```
A manager with 14 direct reports.
```

The lower block will output:

```
Steven King is a manager with 14 direct reports.
```

And this function has the added value of being usable in SQL queries:

Demo 3.6: Conditional-Processing/Demos/num-reports-in-sql-query.sql

```
1.   SELECT employee_id, first_name, last_name,
2.     num_reports(employee_id) AS direct_reports
3.   FROM employees
4.   ORDER BY direct_reports DESC;
```

Code Explanation

The first 10 rows of the results are shown below:

	EMPLOYEE_ID	FIRST_NAME	LAST_NAME	DIRECT_REPORTS
1	100	Steven	King	14
2	123	Shanta	Vollman	8
3	122	Payam	Kaufling	8
4	121	Adam	Fripp	8
5	120	Matthew	Weiss	8
6	124	Kevin	Mourgos	8
7	147	Alberto	Errazuriz	6
8	148	Gerald	Cambrault	6
9	145	John	Russell	6
10	146	Karen	Partners	6

3.5. The CASE Statement

You should be familiar with the two types of `CASE` statements in standard SQL queries:

1. Simple or Selected Case
2. Searched Case

EVALUATION COPY: Not to be used in class.

We can use either one combined with the `num_reports()` function to output `Manager` for managers and `Non-manager` for non-managers:

Demo 3.7: Conditional-Processing/Demos/num-reports-and-sql-case.sql

```
1.   -- Selected Case
2.   SELECT employee_id, first_name, last_name,
3.     CASE num_reports(employee_id)
4.       WHEN 0 THEN 'non-manager'
5.       ELSE 'manager'
6.     END AS is_manager
7.   FROM employees;
8.
9.   -- Searched Case
10.  SELECT employee_id, first_name, last_name,
11.    CASE
12.      WHEN num_reports(employee_id) > 1 THEN 'manager'
13.      ELSE 'non-manager'
14.    END AS is_manager
15.  FROM employees;
```

Evaluation Copy

Code Explanation

The two queries will return the same results. The first 10 rows of the results are shown below:

	EMPLOYEE_ID	FIRST_NAME	LAST_NAME	IS_MANAGER
1	100	Steven	King	manager
2	101	Neena	Kochhar	manager
3	102	Lex	De Haan	manager
4	103	Alexander	Hunold	manager
5	104	Bruce	Ernst	non-manager
6	105	David	Austin	non-manager
7	106	Valli	Pataballa	non-manager
8	107	Diana	Lorentz	non-manager
9	108	Nancy	Greenberg	manager
10	109	Daniel	Faviet	non-manager

PL/SQL also has `CASE` statements, which are used to decide which lines of code to execute.

❖ 3.5.1. Selected Case

```
CASE expression
  WHEN value_1 THEN
    -- statements to execute
  WHEN value_2 THEN
    -- statements to execute
  WHEN value_3 THEN
    -- statements to execute
  ELSE
    -- statements to execute
END CASE;
```

❖ 3.5.2. Searched Case


```
CASE
  WHEN expression_1 THEN
    -- statements to execute
  WHEN expression_2 THEN
    -- statements to execute
  WHEN expression_3 THEN
    -- statements to execute
  ELSE
    -- statements to execute
END CASE;
```

Notice that `CASE` statements end with `END CASE`.

The function in the demo below uses a `CASE` statement to determine whether to return the manager of a department or of an employee:

EVALUATION COPY: Not to be used in class.

Demo 3.8: Conditional-Processing/Demos/get-manager-of-dpt-or-emp.sql

```
1.   -- Create function
2.   CREATE OR REPLACE FUNCTION get_manager_of_dpt_or_emp(
3.     id_in         IN    VARCHAR2,
4.     key_in        IN    VARCHAR2
5.   )
6.    RETURN VARCHAR2
7.   IS
8.     l_manager_name  VARCHAR2(50);
9.   BEGIN
10.    CASE key_in
11.      WHEN 'emp' THEN
12.        SELECT first_name || ' ' || last_name
13.        INTO l_manager_name
14.        FROM employees
15.        WHERE employee_id = (
16.          SELECT manager_id
17.          FROM employees
18.          WHERE employee_id = id_in);
19.      WHEN 'dpt' THEN
20.        SELECT first_name || ' ' || last_name
21.        INTO l_manager_name
22.        FROM employees
23.        WHERE employee_id = (
24.          SELECT manager_id
25.          FROM departments
26.          WHERE manager_id = id_in);
27.      ELSE
28.        l_manager_name := NULL;
29.    END CASE;
30.
31.    RETURN l_manager_name;
32.  END get_manager_of_dpt_or_emp;
33.
34.  -- Call function
35.  SELECT get_manager_of_dpt_or_emp(110, 'emp') FROM DUAL;
36.  SELECT get_manager_of_dpt_or_emp(200, 'dpt') FROM DUAL;
```

Evaluation Copy

3.6. CASE Expressions

PL/SQL also includes `CASE` *expressions*, which are very similar to `CASE` *statements*. The difference is that `CASE` expressions return a single value; whereas, `CASE` statements result in the execution of one or more statements. The result of a `CASE` expression is usually assigned to a variable:

❖ 3.6.1. Selected Case

```
my_var :=
CASE expression
  WHEN value_1 THEN return_value_1
  WHEN value_2 THEN return_value_2
  WHEN value_3 THEN return_value_3
  ELSE return_value_default
END;
```

Evaluation Copy

❖ 3.6.2. Searched Case

```
my_var :=
CASE
  WHEN expression_1 THEN return_value_1
  WHEN expression_2 THEN return_value_2
  WHEN expression_3 THEN return_value_3
  ELSE return_value_default
END;
```

Remember our `is_manager()` function that returns a `BOOLEAN` and how we cannot use it in SQL statements. Examine the code below:

EVALUATION COPY: Not to be used in class.

Demo 3.9: Conditional-Processing/Demos/bool-to-num.sql

```
1.  CREATE OR REPLACE FUNCTION bool_to_num(
2.    bool_in IN BOOLEAN
3.  )
4.   RETURN NUMBER
5.  IS
6.    l_num NUMBER(1,0) := NULL;
7.  BEGIN
8.    l_num := CASE
9.      WHEN bool_in THEN 1
10.     WHEN NOT bool_in THEN 0
11.   END;
12.
13.   RETURN l_num;
14. END bool_to_num;
15.
16. DECLARE
17.   t BOOLEAN := TRUE;
18.   f BOOLEAN := FALSE;
19. BEGIN
20.   DBMS_OUTPUT.PUT_LINE(bool_to_num(t)); -- 1
21.   DBMS_OUTPUT.PUT_LINE(bool_to_num(f)); -- 0
22. END;
```

Evaluation Copy

Code Explanation

The `l_num` variable (for `NUMBER`) will get `1` or `0` depending on whether `bool_in` (for `BOOLEAN`) is `TRUE` or `FALSE`.

`bool_to_num(TRUE)` will return `1`.

`bool_to_num(FALSE)` will return `0`.

We can use the `bool_to_num()` function to quickly create SQL-friendly functions from functions that return `BOOLEAN` values. Here's how we do that with the `is_manager()` function:

Demo 3.10: Conditional-Processing/Demos/is-manager-sql.sql

```
1.   CREATE OR REPLACE FUNCTION is_manager_sql(
2.     employee_id_in IN employees.employee_id%TYPE   := NULL,
3.     first_name_in  IN employees.first_name%TYPE    := NULL,
4.     last_name_in   IN employees.last_name%TYPE     := NULL
5.   )
6.    RETURN NUMBER
7.   IS
8.   BEGIN
9.     RETURN  bool_to_num(
10.              is_manager(
11.                employee_id_in,
12.                first_name_in,
13.                last_name_in
14.              )
15.            );
16.  END is_manager_sql;
```

Evaluation Copy

Code Explanation

The `is_manager_sql()` function takes the same parameters as the `is_manager()` function, but it returns a `NUMBER` instead of a `BOOLEAN`. It then simply makes a call to the `is_manager()` function passing the same parameters it received, and converts the returned `BOOLEAN` value to a `NUMBER` using the `bool_to_num()` function we created.

Now we can use the `is_manager_sql()` function in a SQL query:

Demo 3.11: Conditional-Processing/Demos/is-manager-sql-in-sql-query.sql

```
1.   SELECT employee_id, first_name, last_name,
2.     is_manager_sql(employee_id) AS is_manager
3.   FROM employees;
```


EVALUATION COPY: Not to be used in class.

Exercise 7: Replacing the Head Honcho

20 to 30 minutes

Evaluation Copy

In this exercise you will use `CASE` to decide what to do within a procedure for replacing a head honcho (an employee whose `manager_id` is `NULL`) with another employee. The starting code is shown below:

Exercise Code 7.1: Conditional-Processing/Exercises/replace-head-honcho.sql

```
1.    -- Create the procedure
2.    CREATE OR REPLACE PROCEDURE replace_head_honcho(
3.      new_head_honcho_id_in      IN    employees.employee_id%TYPE,
4.      old_head_honcho_id_in      IN    employees.employee_id%TYPE
5.    ) IS
6.      l_new_head_manager_id   employees.manager_id%TYPE;
7.      l_old_head_manager_id   employees.manager_id%TYPE;
8.      l_new_head_honcho_name  VARCHAR2(50);
9.      l_old_head_honcho_name  VARCHAR2(50);
10.   BEGIN
11.
12.     SELECT manager_id
13.     INTO l_new_head_manager_id
14.     FROM employees
15.     WHERE employee_id = new_head_honcho_id_in;
16.
17.     SELECT manager_id
18.     INTO l_old_head_manager_id
19.     FROM employees
20.     WHERE employee_id = old_head_honcho_id_in;
21.
22.     SELECT first_name || ' ' || last_name
23.     INTO l_new_head_honcho_name
24.     FROM employees
25.     WHERE employee_id = new_head_honcho_id_in;
26.
27.     SELECT first_name || ' ' || last_name
28.     INTO l_old_head_honcho_name
29.     FROM employees
30.     WHERE employee_id = old_head_honcho_id_in;
31.
32.     -- when the new head manager is already a head manager
33.         DBMS_OUTPUT.PUT_LINE(l_new_head_honcho_name ||
34.                              '(employee_id: ' || new_head_honcho_id_in ||
35.                              ') is already a head honcho.');
36.
37.     -- when the old head manager is not actually a head manager
38.         DBMS_OUTPUT.PUT_LINE(l_old_head_honcho_name ||
39.                              '(employee_id: ' || old_head_honcho_id_in ||
40.                              ') is not a head honcho.');
41.     -- otherwise
42.         -- New head honcho should not report to anyone
43.         UPDATE employees
```

Evaluation Copy

EVALUATION COPY: Not to be used in class.

```
44.         SET manager_id = NULL
45.         WHERE employee_id = new_head_honcho_id_in;
46.
47.         -- Old head honcho should report to new head honcho
48.         UPDATE employees
49.         SET manager_id = new_head_honcho_id_in
50.         WHERE employee_id = old_head_honcho_id_in;
51.
52.         DBMS_OUTPUT.PUT_LINE(l_old_head_honcho_name ||
53.                              ' now reports to ' ||
54.                              l_new_head_honcho_name || '.');
55.
56.         DBMS_OUTPUT.PUT_LINE(l_new_head_honcho_name ||
57.                              ' is new head honcho.');
58.
59.
60.   END replace_head_honcho;
61.
62.   -- Call the procedure
63.   DECLARE
64.     l_new_head_honcho_id     employees.employee_id%TYPE   :=  102;
65.     l_old_head_honcho_id     employees.manager_id%TYPE    :=  100;
66.   BEGIN
67.     replace_head_honcho(
68.       new_head_honcho_id_in  => l_new_head_honcho_id,
69.       old_head_honcho_id_in  => l_old_head_honcho_id
70.     );
71.   END;
```

Evaluation Copy

Code Explanation

The procedure includes:

1. Two `IN` parameters: the employee ids of the employee becoming a head honcho and of the employee who is losing their head honcho status.
2. Four local variables for storing the employees' full names and their managers' ids.
3. Four `SELECT...INTO` statements for setting the values of the local variables.
4. Two `UPDATE` statements for setting the new `manager_id` values of the passed-in employees.

Open `Conditional-Processing/Exercises/replace-head-honcho.sql` in SQL Developer.

1. Your job is to use a `CASE` statement to branch the code below the four `SELECT` statements and before the end of the procedure. Use the comments in the code for guidance.
2. When you have finished, run the procedure code to create the procedure. If you get any errors, fix them and run it again until it compiles without errors.
3. Now test your code:
    A. Run the block of code below the procedure. It should output:

```
Steven King now reports to Lex De Haan.
Lex De Haan is new head honcho.
```

    B. Run the block of code again without making any changes. This time, it should output:

Evaluation Copy

```
Lex De Haan(employee_id: 102) is already a head honcho.
```

    C. Now change the values of the `l_new_head_honcho_id` and `l_old_head_honcho_id` to `103` and `104`, respectively. Neither of these employees is currently a head honcho. Run the block again. It should output:

```
Bruce Ernst(employee_id: 104) is not a head honcho.
```


EVALUATION COPY: Not to be used in class.

Solution: Conditional-Processing/Solutions/replace-head-honcho.sql

```
1.   -- Create the procedure
2.   CREATE OR REPLACE PROCEDURE replace_head_honcho(
3.     new_head_honcho_id_in      IN    employees.employee_id%TYPE,
4.     old_head_honcho_id_in      IN    employees.employee_id%TYPE
5.   ) IS
6.     l_new_head_manager_id   employees.manager_id%TYPE;
7.     l_old_head_manager_id   employees.manager_id%TYPE;
8.     l_new_head_honcho_name  VARCHAR2(50);
9.     l_old_head_honcho_name  VARCHAR2(50);
10.  BEGIN
11.
12.    SELECT manager_id
13.    INTO l_new_head_manager_id
14.    FROM employees
15.    WHERE employee_id = new_head_honcho_id_in;
16.
17.    SELECT manager_id
18.    INTO l_old_head_manager_id
19.    FROM employees
20.    WHERE employee_id = old_head_honcho_id_in;
21.
22.    SELECT first_name || ' ' || last_name
23.    INTO l_new_head_honcho_name
24.    FROM employees
25.    WHERE employee_id = new_head_honcho_id_in;
26.
27.    SELECT first_name || ' ' || last_name
28.    INTO l_old_head_honcho_name
29.    FROM employees
30.    WHERE employee_id = old_head_honcho_id_in;
31.
32.    CASE
33.      WHEN l_new_head_manager_id IS NULL THEN
34.        DBMS_OUTPUT.PUT_LINE(l_new_head_honcho_name ||
35.                             '(employee_id: ' || new_head_honcho_id_in ||
36.                             ') is already a head honcho.');
37.      WHEN l_old_head_manager_id IS NOT NULL THEN
38.        DBMS_OUTPUT.PUT_LINE(l_old_head_honcho_name ||
39.                             '(employee_id: ' || old_head_honcho_id_in ||
40.                             ') is not a head honcho.');
41.      ELSE
42.        -- New head honcho should not report to anyone
43.        UPDATE employees
44.        SET manager_id = NULL
```

Evaluation Copy

EVALUATION COPY: Not to be used in class.

```
45.         WHERE employee_id = new_head_honcho_id_in;
46.
47.         -- Old head honcho should report to new head honcho
48.         UPDATE employees
49.         SET manager_id = new_head_honcho_id_in
50.         WHERE employee_id = old_head_honcho_id_in;
51.
52.         DBMS_OUTPUT.PUT_LINE(l_old_head_honcho_name ||
53.                              ' now reports to ' ||
54.                              l_new_head_honcho_name || '.');
55.
56.         DBMS_OUTPUT.PUT_LINE(l_new_head_honcho_name ||
57.                              ' is new head honcho.');
58.     END CASE;
59.
60.   END replace_head_honcho;
61.
62.   -- Call the procedure
63.   DECLARE
64.     l_new_head_honcho_id     employees.employee_id%TYPE  :=  102;
65.     l_old_head_honcho_id     employees.manager_id%TYPE   :=  100;
66.   BEGIN
67.     replace_head_honcho(
68.       new_head_honcho_id_in  => l_new_head_honcho_id,
69.       old_head_honcho_id_in  => l_old_head_honcho_id
70.     );
71.   END;
```

Evaluation Copy

Code Explanation

In practice, we would not output messages for each condition in the `CASE` statement within the procedure. Instead we would raise exceptions for the first two conditions and perhaps set `OUT` parameters for storing the full names of the two employees, so that the calling code could output messages if necessary. We will do this is the lesson on Exceptions.

Conclusion

In this lesson, you have learned about `BOOLEAN` values and expressions and to write PL/SQL conditional statements using `IF` conditions and `CASE` statements.

EVALUATION COPY: Not to be used in class.

LESSON 4
Exceptions

Topics Covered

☑ Exceptions.

Introduction

In this lesson, you will learn to handle exceptions in PL/SQL.

4.1. Introduction to Exceptions

Evaluation Copy

By this time, you may have run into some exceptions (runtime errors) in your PL/SQL code. Often you will want to anticipate, catch, and handle these exceptions in some special way. You do this in PL/SQL in the `EXCEPTION` block. Consider the following code:

Demo 4.1: Exceptions/Demos/block.sql

```
1.   DECLARE
2.     l_greeting VARCHAR2(50);
3.   BEGIN
4.     SELECT 'Hello, ' || region_name || '!'
5.     INTO l_greeting
6.     FROM regions;
7.
8.     DBMS_OUTPUT.PUT_LINE(l_greeting);
9.   END;
```

Code Explanation

Run this and you will get an error similar to the one below:

```
Error report -
ORA-01422: exact fetch returns more than requested number of rows
ORA-06512: at line 4
01422. 00000 - "exact fetch returns more than requested number of rows"
*Cause:    The number specified in exact fetch is less than the rows returned.
*Action:   Rewrite the query or change number of rows requested
```

The problem here is that the `SELECT` statement returns multiple rows, but we can only select one value into the `l_greeting` variable.

This is an example of an uncaught exception. Uncaught exceptions return ugly error messages that are intended to be useful to the developer, but are not useful for the end user. In this lesson, we will learn to catch these exceptions and either output useful information or work around the exception in some way.

Evaluation Copy

ORA-06512

The `ORA-06512` error code simply indicates that there is an exception at the indicated line. It's usually the exception shown before that that you need to resolve. In the error report above, the important message relates to the `ORA-01422` error code.

EVALUATION COPY: Not to be used in class.

4.2. Predefined Exceptions

Oracle gives names to the most common internally defined exceptions. These names are listed in the table below:

Exception Name	Oracle Error Code	ANSI SQL Error Code
ACCESS_INTO_NULL	ORA-06530	-6530
CASE_NOT_FOUND	ORA-06592	-6592
COLLECTION_IS_NULL	ORA-06531	-6531
CURSOR_ALREADY_OPEN	ORA-06511	-6511
DUP_VAL_ON_INDEX	ORA-00001	-1
INVALID_CURSOR	ORA-01001	-1001
INVALID_NUMBER	ORA-01722	-1722
LOGIN_DENIED	ORA-01017	-1017
NO_DATA_FOUND	ORA-01403	+100
NO_DATA_NEEDED	ORA-06548	-6548
NOT_LOGGED_ON	ORA-01012	-1012
PROGRAM_ERROR	ORA-06501	-6501
ROWTYPE_MISMATCH	ORA-06504	-6504
SELF_IS_NULL	ORA-30625	-30625
STORAGE_ERROR	ORA-06500	-6500
SUBSCRIPT_BEYOND_COUNT	ORA-06533	-6533
SUBSCRIPT_OUTSIDE_LIMIT	ORA-06532	-6532
SYS_INVALID_ROWID	ORA-01410	-1410
TIMEOUT_ON_RESOURCE	ORA-00051	-51
TOO_MANY_ROWS	ORA-01422	-1422
VALUE_ERROR	ORA-06502	-6502
ZERO_DIVIDE	ORA-01476	-1476

Evaluation Copy

Note that, with one exception (*no pun intended*), the Oracle-specific codes are the same as the ANSI SQL codes, but with an "ORA" prefix and leading zeroes to make the code five digits long. The exception to this is the `NO_DATA_FOUND` exception, which has an Oracle error code of `ORA-01403` and an ANSI SQL error code of `+100`.

This list is available in the Oracle documentation[3] under **Predefined Exceptions**.

4.3. The EXCEPTION Part of the Block

The syntax for a block with an `EXCEPTION` part is as follows:

```
DECLARE
  /* declarations */
BEGIN
  /* executable commands */
EXCEPTION
  /* error handling code */
END;
```

Evaluation Copy

The syntax of the `EXCEPTION` part is similar to that of a searched `CASE`:

```
EXCEPTION
  WHEN EXCEPTION_NAME_1 THEN
    -- Do something 1
  WHEN EXCEPTION_NAME_2 THEN
    -- Do something 2
  WHEN EXCEPTION_NAME_3 THEN
    -- Do something 3
  WHEN OTHERS THEN
    -- Do something 4
```

The demo below shows the same block we saw earlier that caused an exception, except this time, we catch the exception and output a friendlier error message:

3. https://docs.oracle.com/en/database/oracle/oracle-database/19/lnpls/plsql-error-handling.html

EVALUATION COPY: Not to be used in class.

Demo 4.2: Exceptions/Demos/too-many-rows.sql

```
1.   DECLARE
2.     l_greeting VARCHAR2(50);
3.   BEGIN
4.     SELECT 'Hello, ' || region_name || '!'
5.     INTO l_greeting
6.     FROM regions;
7.
8.     DBMS_OUTPUT.PUT_LINE(l_greeting);
9.   EXCEPTION
10.    WHEN TOO_MANY_ROWS THEN
11.      DBMS_OUTPUT.PUT_LINE('ERROR: Query returns more than one row.');
12.  END;
```

Evaluation Copy

Code Explanation

This will simply output:

```
ERROR: Query returns more than one row.
```

Note that this code will only catch a `TOO_MANY_ROWS` exception. Any other type of exception will result in the same ugly error messages we saw before.

In the next demo, we add a `WHEN OTHERS THEN` clause, which catches all other types of errors:

Demo 4.3: Exceptions/Demos/when-others.sql

```
1.   DECLARE
2.     l_greeting VARCHAR2(50);
3.   BEGIN
4.     SELECT 'Hello, ' || region_name || '!'
5.     INTO l_greeting
6.     FROM regions;
7.
8.     DBMS_OUTPUT.PUT_LINE(l_greeting);
9.   EXCEPTION
10.    WHEN TOO_MANY_ROWS THEN
11.      DBMS_OUTPUT.PUT_LINE('ERROR: Query returns more than one row.');
12.    WHEN OTHERS THEN
13.      RAISE_APPLICATION_ERROR(-20001, 'An error was encountered - ' ||
14.                                      SQLCODE ||
15.                                      ' -ERROR- ' ||
16.                                      SQLERRM);
17.  END;
```

Code Explanation

Evaluation Copy

We first catch and handle the `TOO_MANY_ROWS` exception. Then, we catch any other exception that occurs and we use the built-in `RAISE_APPLICATION_ERROR()` stored procedure and the `SQLCODE` and `SQLERRM` functions to output the ANSI SQL error code and the error message, which contains the Oracle error code. We will explain `RAISE_APPLICATION_ERROR()` in more detail soon.

Using the information provided by the `SQLCODE` and `SQLERRM` functions, we can improve our code to explicitly catch those exceptions as well. To illustrate this, open `Exceptions/Demos/when-others.sql` in SQL Developer and add the `WHERE` clause shown below:

```
SELECT 'Hello, ' || region_name || '!'
INTO l_greeting
FROM regions
WHERE region_id = 100;
```

Run the code. You should see an error like this one:

```
Error report -
ORA-20001: An error was encountered - 100 -ERROR- ORA-01403: no data found
ORA-06512: at line 14
```


EVALUATION COPY: Not to be used in class.

Note the two error codes:

1. `100` - The ANSI SQL error code.
2. `ORA-01403` - The Oracle error code.

Evaluation Copy

We can use these to look up the exception name, which is `NO_DATA_FOUND`. The problem here is that the `SELECT` statement doesn't return any rows, but we must select one (and only one) value into the `l_greeting` variable.

In the following exercise, you'll add code to catch `NO_DATA_FOUND` exceptions.

Exercise 8: Catching NO_DATA_FOUND Exception

5 to 10 minutes

Evaluation Copy

In this exercise you will modify the code from the previous demo to catch the `NO_DATA_FOUND` exception as well as the `TOO_MANY_ROWS` exception.

1. Open `Exceptions/Exercises/no-data-found.sql` in SQL Developer.
2. Add code to catch the `NO_DATA_FOUND` exception and output a friendly error message.

EVALUATION COPY: Not to be used in class.

Solution: Exceptions/Solutions/no-data-found.sql

```
1.  DECLARE
2.    l_greeting VARCHAR2(50);
3.  BEGIN
4.    SELECT 'Hello, ' || region_name || '!'
5.    INTO l_greeting
6.    FROM regions
7.    WHERE region_id = 100;
8.
9.    DBMS_OUTPUT.PUT_LINE(l_greeting);
10. EXCEPTION
11.   WHEN TOO_MANY_ROWS THEN
12.     DBMS_OUTPUT.PUT_LINE('ERROR: Query returns more than one row.');
13.   WHEN NO_DATA_FOUND THEN
14.     DBMS_OUTPUT.PUT_LINE('ERROR: Query must return a row.');
15.   WHEN OTHERS THEN
16.     RAISE_APPLICATION_ERROR(-20001,'An error was encountered - ' ||
17.                             SQLCODE ||
18.                             ' -ERROR- ' ||
19.                             SQLERRM);
20. END;
```

Evaluation Copy

4.4. User-defined Exceptions

In addition to those that are predefined, you can define your own exceptions:

```
DECLARE
  exception_name EXCEPTION;
```

The demo below shows a simple user-defined exception:

EVALUATION COPY: Not to be used in class.

Demo 4.4: Exceptions/Demos/user-defined-exception-simple.sql

```
1.   DECLARE
2.     l_employee_id       employees.employee_id%TYPE := 101;
3.     l_manager_id        employees.manager_id%TYPE  := 101;
4.     manager_is_self     EXCEPTION;
5.   BEGIN
6.     IF l_employee_id = l_manager_id THEN
7.       RAISE manager_is_self;
8.     END IF;
9.
10.    UPDATE employees
11.    SET manager_id = l_manager_id
12.    WHERE employee_id = l_employee_id;
13.
14.  EXCEPTION
15.    WHEN manager_is_self THEN
16.      DBMS_OUTPUT.PUT_LINE('Employee cannot be own manager.');
17.    WHEN OTHERS THEN
18.      RAISE_APPLICATION_ERROR(-20001,'An error was encountered - ' ||
19.                                     SQLCODE ||
20.                                     ' -ERROR- ' ||
21.                                     SQLERRM);
22.  END;
```

Evaluation Copy

Code Explanation

The HR database doesn't prevent employees from being their own managers. For example, it would allow this:

```
UPDATE employees
SET manager_id = 204
WHERE employee_id = 204;
```

That doesn't make much sense though. There is only one employee who manages themself: Steven King. And his `manager_id` is set to `NULL` to indicate that he doesn't have a manager.

Our code throws a `manager_is_self` exception when you try to set the `manager_id` to the same value as the `employee_id`.

4.5. User-defined Exceptions in Subprograms

It is common to include exception handling in subprograms. The following code turns the previous code into a procedure, so that it can be called like this:

```
update_employee_manager(204, 100);
```

Or using named notation, like this:


```
update_employee_manager(
  employee_id_in     => 204,
  new_manager_id_in  => 100
);
```


EVALUATION COPY: Not to be used in class.

Demo 4.5: Exceptions/Demos/update-employee-manager-1.sql

```
1.   -- Create the procedure
2.   CREATE OR REPLACE PROCEDURE update_employee_manager(
3.     employee_id_in    IN  employees.employee_id%TYPE,
4.     new_manager_id_in IN  employees.manager_id%TYPE
5.   ) IS
6.     manager_is_self      EXCEPTION;
7.   BEGIN
8.     IF employee_id_in = new_manager_id_in THEN
9.       RAISE manager_is_self;
10.    END IF;
11.
12.    -- Set new manager
13.    UPDATE employees
14.    SET manager_id = new_manager_id_in
15.    WHERE employee_id = employee_id_in;
16.    DBMS_OUTPUT.PUT_LINE('Success: Employee manager updated.');
17.
18.  EXCEPTION
19.    WHEN manager_is_self THEN
20.      DBMS_OUTPUT.PUT_LINE('Error: Employee cannot be own manager.');
21.  END update_employee_manager;
22.
23.  -- Call the procedure
24.  DECLARE
25.    l_employee_id  employees.employee_id%TYPE  :=  101;
26.    l_manager_id   employees.manager_id%TYPE   :=  101;
27.
28.  BEGIN
29.    update_employee_manager(
30.      employee_id_in     => l_employee_id,
31.      new_manager_id_in  => l_manager_id
32.    );
33.  END;
```

Evaluation Copy

Code Explanation

Open `Exceptions/Demos/update-employee-manager-1.sql` in SQL Developer and try the following:

1. Run the code to create the `update_employee_manager()` procedure.

2. Run the block of code below the procedure to call it. Notice we're passing in `101` for both the `employee_id` and the `new_manager_id`. This will result in the following output:

```
Error: Employee cannot be own manager.
```

3. Now change `l_employee_id` to `204` and `l_manager_id` to `100`:

```
DECLARE
      l_employee_id   employees.employee_id%TYPE   :=  204;
      l_manager_id    employees.manager_id%TYPE    :=  100;
```

And then run that block of code again. This time it will not error, and it will output:

```
Success: Employee manager updated.
```

4. Execute a `ROLLBACK` to undo these changes:

```
ROLLBACK;
```

Evaluation Copy

4.6. Re-raising Exceptions

In the `Exceptions/Demos/update-employee-manager-1.sql` demo we just looked at, we used `DBMS_OUTPUT.PUT_LINE()` in the procedure to output an error message. A better approach would be to re-raise the exception and let the calling code decide whether or not to output a message. There are a couple of ways to re-raise exceptions. The simplest is to just call `RAISE`:

```
EXCEPTION
  WHEN manager_is_self THEN
    RAISE;
```

`RAISE` simply re-raises the current exception so that the calling code can catch and handle it. The downside is that the calling code gets little information about the exception.

Usually, a better option is to use the `RAISE_APPLICATION_ERROR()` stored procedure, which takes three arguments:

EVALUATION COPY: Not to be used in class.

1. An error code in the range `-20000` up to `-20999`. Oracle reserves this range of negative integers for use by application developers.
2. An error message.
3. An optional Boolean (`TRUE` or `FALSE`) indicating whether or not to show the *error stack*: the errors leading up to the error that was caught. The default is `FALSE`, but showing the error stack can be helpful for debugging your code.

So, instead of outputting a message like this:

```
WHEN manager_is_self THEN
  DBMS_OUTPUT.PUT_LINE('Error: Employee cannot be own manager.');
```

We re-raise the error like this:

Evaluation Copy

```
EXCEPTION
  WHEN manager_is_self THEN
    RAISE_APPLICATION_ERROR(-20101,
                            'Employee cannot be own manager.');
```

Now the calling code can catch the error and output some information about it:

```
EXCEPTION
  WHEN OTHERS THEN
    RAISE_APPLICATION_ERROR(-20001, 'An error was encountered - ' ||
                                    SQLCODE ||
                                    ' -ERROR- ' ||
                                    SQLERRM, TRUE);
```

These changes are included in the following demo:

Demo 4.6: Exceptions/Demos/update-employee-manager-2.sql

```
-------Lines 1 through 6 Omitted-------
BEGIN
  IF employee_id_in = new_manager_id_in THEN
    RAISE manager_is_self;
  END IF;

  -- Set new manager
  UPDATE employees
  SET manager_id = new_manager_id_in
  WHERE employee_id = employee_id_in;
  DBMS_OUTPUT.PUT_LINE('Success: Employee manager updated.');

EXCEPTION
  WHEN manager_is_self THEN
    -- re-raise the error to be caught by calling code
    RAISE_APPLICATION_ERROR(-20101,
                            'Employee cannot be own manager.');
END update_employee_manager;

-- Call the procedure
DECLARE
  l_employee_id   employees.employee_id%TYPE  :=  101;
  l_manager_id    employees.manager_id%TYPE  :=  101;
BEGIN
  update_employee_manager(
    employee_id_in    => l_employee_id,
    new_manager_id_in => l_manager_id
  );
EXCEPTION
  WHEN OTHERS THEN
    RAISE_APPLICATION_ERROR(-20001, 'An error was encountered - ' ||
                                    SQLCODE ||
                                    ' -ERROR- ' ||
                                    SQLERRM, TRUE);
END;
```

Evaluation Copy

Code Explanation

Run this code. You should get an error similar to the following:

EVALUATION COPY: Not to be used in class.

```
Error report -
ORA-20001: An error was encountered - -20101 -ERROR- ORA-20101: Employee cannot be own
manager.
ORA-06512: at line 11
ORA-20101: Employee cannot be own manager.
ORA-06512: at "HR.UPDATE_EMPLOYEE_MANAGER", line 20
ORA-06512: at line 5
```

Evaluation Copy

Notice how helpful the error stack is. The most recent error is shown first and you can follow the trace backward. The `ORA-06512` error codes provide line numbers. Note that these line numbers are not equivalent to file line numbers. Their starting point is not the beginning of the file, but the beginning of the block or subprogram in which the error occurred.

Exercise 9: Replacing the Head Honcho (revisited)

Evaluation Copy

20 to 30 minutes

Earlier, we created a procedure for replacing a head honcho. The code is shown below. Please review it here or in SQL Developer:

EVALUATION COPY: Not to be used in class.

Exercise Code 9.1: Exceptions/Exercises/replace-head-honcho.sql

```
1.   -- Create the procedure
2.   CREATE OR REPLACE PROCEDURE replace_head_honcho(
3.     new_head_honcho_id_in      IN    employees.employee_id%TYPE,
4.     old_head_honcho_id_in      IN    employees.employee_id%TYPE
5.   ) IS
6.     l_new_head_manager_id   employees.manager_id%TYPE;
7.     l_old_head_manager_id   employees.manager_id%TYPE;
8.     l_new_head_honcho_name  VARCHAR2(50);
9.     l_old_head_honcho_name  VARCHAR2(50);
10.  BEGIN
11.
12.    SELECT manager_id
13.    INTO l_new_head_manager_id
14.    FROM employees
15.    WHERE employee_id = new_head_honcho_id_in;
16.
17.    SELECT manager_id
18.    INTO l_old_head_manager_id
19.    FROM employees
20.    WHERE employee_id = old_head_honcho_id_in;
21.
22.    SELECT first_name || ' ' || last_name
23.    INTO l_new_head_honcho_name
24.    FROM employees
25.    WHERE employee_id = new_head_honcho_id_in;
26.
27.    SELECT first_name || ' ' || last_name
28.    INTO l_old_head_honcho_name
29.    FROM employees
30.    WHERE employee_id = old_head_honcho_id_in;
31.
32.    CASE
33.      WHEN l_new_head_manager_id IS NULL THEN
34.        DBMS_OUTPUT.PUT_LINE(l_new_head_honcho_name ||
35.                             '(employee_id: ' || new_head_honcho_id_in ||
36.                             ') is already a head honcho.');
37.      WHEN l_old_head_manager_id IS NOT NULL THEN
38.        DBMS_OUTPUT.PUT_LINE(l_old_head_honcho_name ||
39.                             '(employee_id: ' || old_head_honcho_id_in ||
40.                             ') is not a head honcho.');
41.      ELSE
42.        -- New head honcho should not report to anyone
43.        UPDATE employees
44.        SET manager_id = NULL
```

Evaluation Copy

```
45.         WHERE employee_id = new_head_honcho_id_in;
46.
47.         -- Old head honcho should report to new head honcho
48.         UPDATE employees
49.         SET manager_id = new_head_honcho_id_in
50.         WHERE employee_id = old_head_honcho_id_in;
51.
52.         DBMS_OUTPUT.PUT_LINE(l_old_head_honcho_name ||
53.                              ' now reports to ' ||
54.                              l_new_head_honcho_name || '.');
55.
56.         DBMS_OUTPUT.PUT_LINE(l_new_head_honcho_name ||
57.                              ' is new head honcho.');
58.     END CASE;
59.
60.   END replace_head_honcho;
61.
      -------Lines 62 through 88 Omitted-------
```

Evaluation Copy

In this exercise, you will replace the output messages with exceptions and write code to handle those exceptions.

1. Open `Exceptions/Exercises/replace-head-honcho.sql` in SQL Developer.
2. Remove the local variables for storing the employees' names and instead add two `OUT` parameters for the same purpose.
3. Update the two last `SELECT...INTO` statements to select into the two `OUT` parameters you just added.
4. Replace all other uses of employee name local variables (e.g., `l_old_head_honcho_name`) with the new `OUT` parameters.
5. Declare two exceptions: `employee_already_head_honcho` and `employee_not_head_honcho` and replace the output in the `WHEN` conditions of the `CASE` statement with code raising these exceptions.
6. Add an `EXCEPTION` part to the procedure for handling the two exceptions. The error messages should be similar to the original output messages.
7. Remove the output in the `ELSE` part of the `CASE` statement.
8. Run the code to create the procedure and test your code:

EVALUATION COPY: Not to be used in class.

A. Note that the block below the procedure has already been rewritten to assume the changes that you have made above. Run the block. It should output:

```
Steven King now reports to Lex De Haan.
Lex De Haan is new head honcho.
```

You may get different output due to changes you have already made in the `employees` table. If you do, try switching the numbers.

B. Run the block of code again without making any changes. This time, it should output:

```
Lex De Haan(employee_id: 102) is already a head honcho.
```

Evaluation Copy

C. Now change the values of the `l_new_head_honcho_id` and `l_old_head_honcho_id` to `103` and `104`, respectively and run it again. It should output:

```
Bruce Ernst(employee_id: 104) is not a head honcho.
```

9. Execute a `ROLLBACK` to undo these changes:

```
ROLLBACK;
```

Solution: Exceptions/Solutions/replace-head-honcho.sql

```
1.  -- Create the procedure
2.  CREATE OR REPLACE PROCEDURE replace_head_honcho(
3.    new_head_honcho_id_in      IN    employees.employee_id%TYPE,
4.    old_head_honcho_id_in      IN    employees.employee_id%TYPE,
5.    new_head_honcho_name_out   OUT   VARCHAR2,
6.    old_head_honcho_name_out   OUT   VARCHAR2
7.  ) IS
8.    l_new_head_manager_id         employees.manager_id%TYPE;
9.    l_old_head_manager_id         employees.manager_id%TYPE;
10.   employee_already_head_honcho  EXCEPTION;
11.   employee_not_head_honcho      EXCEPTION;
12. BEGIN
    -------Lines 13 through 23 Omitted-------
24.   SELECT first_name || ' ' || last_name
25.   INTO new_head_honcho_name_out
26.   FROM employees
27.   WHERE employee_id = new_head_honcho_id_in;
28.
29.   SELECT first_name || ' ' || last_name
30.   INTO old_head_honcho_name_out
31.   FROM employees
32.   WHERE employee_id = old_head_honcho_id_in;
33.
34.   CASE
35.     WHEN l_new_head_manager_id IS NULL THEN
36.       RAISE employee_already_head_honcho;
37.     WHEN l_old_head_manager_id IS NOT NULL THEN
38.       RAISE employee_not_head_honcho;
39.     ELSE
    -------Lines 40 through 49 Omitted-------
50. EXCEPTION
51.   WHEN employee_already_head_honcho THEN
52.     RAISE_APPLICATION_ERROR(-20101,
53.               new_head_honcho_name_out ||
54.               '(employee_id: ' || new_head_honcho_id_in ||
55.               ') is already a head honcho.');
56.   WHEN employee_not_head_honcho THEN
57.     RAISE_APPLICATION_ERROR(-20102,
58.               old_head_honcho_name_out ||
59.               '(employee_id: ' || old_head_honcho_id_in ||
60.               ') is not a head honcho.');
61.
62. END replace_head_honcho;
```

Evaluation Copy

EVALUATION COPY: Not to be used in class.

```
63.
64.  -- Call the procedure
65.  DECLARE
66.    l_new_head_honcho_id    employees.employee_id%TYPE  :=  102;
67.    l_old_head_honcho_id    employees.manager_id%TYPE   :=  100;
68.    l_new_head_honcho_name  VARCHAR2(50);
69.    l_old_head_honcho_name  VARCHAR2(50);
70.  BEGIN
71.    replace_head_honcho(
72.      new_head_honcho_id_in     => l_new_head_honcho_id,
73.      old_head_honcho_id_in     => l_old_head_honcho_id,
74.      new_head_honcho_name_out  => l_new_head_honcho_name,
75.      old_head_honcho_name_out  => l_old_head_honcho_name
76.    );
77.
78.    DBMS_OUTPUT.PUT_LINE(l_old_head_honcho_name ||
79.                         ' now reports to ' ||
80.                         l_new_head_honcho_name || '.');
81.
82.    DBMS_OUTPUT.PUT_LINE(l_new_head_honcho_name ||
83.                         ' is new head honcho.');
84.  EXCEPTION
85.    WHEN OTHERS THEN
86.      RAISE_APPLICATION_ERROR(-20001, 'An error was encountered - ' ||
87.                                      SQLCODE ||
88.                                      ' -ERROR- ' ||
89.                                      SQLERRM, TRUE);
90.  END;
```

Exercise 10: Adding Exceptions to update_employee_manager()

30 to 60 minutes

In this exercise, you will add exception handling to the `update_employee_manager()` procedure to prevent:

1. Reciprocal employee-manager relationships. Employees cannot manage their managers.
2. Removing the last head honcho. There must be at least one top-level boss, whose `manager_id` is `NULL`. Our code already allows you to assign a new head honcho by passing in `NULL` for the manager id of an employee. However, it does not prevent you from assigning a `manager_id` to the last existing head honcho, which would leave us with no top-level leader.

Open `Exceptions/Exercises/update-employee-manager-3.sql` in SQL Developer.

1. Add the following `OUT` parameters:
    A. `employee_name_out` - The full name of the employee matching the passed-in `employee_id_in`.
    B. `old_manager_name_out` - The full name of the current manager of the employee matching the passed-in `employee_id_in`.
    C. `new_manager_name_out` - The full name of the manager matching the passed-in `new_manager_id_in`.

Evaluation Copy

2. In the `IS` part of the procedure, declare the following new variables:
    A. `l_old_manager_id` - to hold employee's current manager's `employee_id` (i.e., the employee's `manager_id`). Should default to `0`.
    B. `l_new_managers_manager_id` - to hold the new manager's `manager_id`; (i.e., the `manager_id` of the new manager). Should default to `0`.
    C. `l_num_head_honchos` - to hold the number of employees who have no manager assigned.
    D. `cannot_remove_last_head_honcho` - Exception.
    E. `reciprocal_managers` - Exception.

EVALUATION COPY: Not to be used in class.

3. Select the employee's current `manager_id` and full name into `l_old_manager_id` and `employee_name_out`, respectively.

4. If the passed-in `new_manager_id_in` is not `NULL`, it means that the employee is being assigned a new manager. Select this new manager's `manager_id` and full name into `l_new_managers_manager_id` and `new_manager_name_out`, respectively. We need to know the new manager's `manager_id` so that we can avoid a reciprocal employee-manager relationship, like the following:

```
employee_id     manager_id
100             110
110             100
```

5. If `l_old_manager_id` is not `NULL`, it means that the employee had a previous manager. Select that manager's name into `old_manager_name_out`. This step is already done for you.

6. If `new_manager_id_in` is not `NULL`, that means we are assigning a new manager to this employee. We have two concerns:

   - That manager cannot be someone who reports to the employee.
   - If this employee is the only head honcho, we can't assign them a manager as that would leave us with no head honchos.

   Evaluation Copy

   These checks have been added already. You just need to raise the appropriate exceptions:

   A. Raise a `reciprocal_managers` exception if `employee_id_in` is equal to `l_new_managers_manager_id`.

   B. Raise a `cannot_remove_last_head_honcho` exception if `l_num_head_honchos` is equal to `0`.

7. When there is a `reciprocal_managers` exception, re-raise the exception with an appropriate error code and message.

8. When there is a `cannot_remove_last_head_honcho` exception, re-raise the exception with an appropriate error code and message.

9. Run the code to create the procedure and test your code with the following variable values:

   A. `l_employee_id`: `204`, `l_new_manager_id`: `100`. This should output:

```
Employee:    Hermann Baer
Old Manager: Neena Kochhar
New Manager: Steven King
```

B. Run the block again without making any changes. It should output:

```
Employee:    Hermann Baer
Old Manager: Steven King
New Manager: Steven King
```

C. `l_employee_id`: `100`, `l_new_manager_id`: `102`. This should output:

```
ORA-20001: An error was encountered - -20102 -ERROR- ORA-20102: Employees
 cannot manage each other.
```

D. `l_employee_id`: `103`, `l_new_manager_id`: `NULL`. This should output:

```
Employee:    Alexander Hunold
Old Manager: Lex De Haan
New Manager:
```

Evaluation Copy

E. `l_employee_id`: `103`, `l_new_manager_id`: `102`. This should output:

```
Employee:    Alexander Hunold
Old Manager:
New Manager: Lex De Haan
```

F. `l_employee_id`: `100`, `l_new_manager_id`: `103`. This should output:

```
ORA-20001: An error was encountered - -20103 -ERROR- ORA-20103: Cannot
remove last head honcho.
```

10. Execute a `ROLLBACK` to undo these changes:

```
ROLLBACK;
```


EVALUATION COPY: Not to be used in class.

Solution: Exceptions/Solutions/update-employee-manager-3.sql

```
1.  -- Create the procedure
2.  CREATE OR REPLACE PROCEDURE update_employee_manager(
3.    employee_id_in        IN    employees.employee_id%TYPE,
4.    new_manager_id_in     IN    employees.manager_id%TYPE,
5.    employee_name_out     OUT   VARCHAR2,
6.    old_manager_name_out  OUT   VARCHAR2,
7.    new_manager_name_out  OUT   VARCHAR2
8.  ) IS
9.    l_old_manager_id               employees.manager_id%TYPE := 0;
10.   l_new_managers_manager_id      employees.manager_id%TYPE := 0;
11.   l_num_head_honchos             NUMBER(6,0);
12.   manager_is_self                EXCEPTION;
13.   cannot_remove_last_head_honcho EXCEPTION;
14.   reciprocal_managers            EXCEPTION;
15. BEGIN
16.   IF employee_id_in = new_manager_id_in THEN
17.     RAISE manager_is_self;
18.   END IF;
19.
20.   -- Get manager id and name of employee
21.   SELECT manager_id, first_name || ' ' || last_name
22.   INTO l_old_manager_id, employee_name_out
23.   FROM employees
24.   WHERE employee_id = employee_id_in;
25.
26.   -- Get name of old manager
27.   IF l_old_manager_id IS NOT NULL THEN
28.     SELECT first_name || ' ' || last_name
29.     INTO old_manager_name_out
30.     FROM employees
31.     WHERE employee_id = l_old_manager_id;
32.   END IF;
33.
34.   IF new_manager_id_in IS NOT NULL THEN
35.     -- Get manager id and full name of the passed-in manager
36.     SELECT manager_id, first_name || ' ' || last_name
37.     INTO l_new_managers_manager_id, new_manager_name_out
38.     FROM employees
39.     WHERE employee_id = new_manager_id_in;
40.
41.     -- A manager cannot report to someone who reports to them
42.     IF employee_id_in = l_new_managers_manager_id THEN
43.       RAISE reciprocal_managers;
44.     END IF;
```


EVALUATION COPY: Not to be used in class.

```
45.
46.       -- Get number of head honchos not including this employee
47.       -- This will be the number of head honchos left after assigning
48.       -- this employee a manager.
49.       SELECT COUNT(employee_id)
50.       INTO l_num_head_honchos
51.       FROM employees
52.       WHERE manager_id IS NULL AND employee_id <> employee_id_in;
53.
54.       -- Cannot remove last head honcho
55.       IF l_num_head_honchos = 0 THEN
56.         RAISE cannot_remove_last_head_honcho;
57.       END IF;
58.     END IF;
59.
60.     -- Set new manager
61.     UPDATE employees
62.     SET manager_id = new_manager_id_in
63.     WHERE employee_id = employee_id_in;
64.
65.   EXCEPTION
66.     WHEN manager_is_self THEN
67.       RAISE_APPLICATION_ERROR(-20101,
68.                               'Employee cannot be own manager.');
69.     WHEN reciprocal_managers THEN
70.       RAISE_APPLICATION_ERROR(-20102,
71.                               'Employees cannot manage each other.');
72.     WHEN cannot_remove_last_head_honcho THEN
73.       RAISE_APPLICATION_ERROR(-20103,
74.                               'Cannot remove last head honcho.');
75.   END update_employee_manager;
76.
77.   -- Call the procedure
78.   DECLARE
79.     l_employee_id       employees.employee_id%TYPE  :=  204;
80.     l_new_manager_id    employees.manager_id%TYPE   :=  100;
81.     l_employee_name     VARCHAR2(50);
82.     l_old_manager_name  VARCHAR2(50);
83.     l_new_manager_name  VARCHAR2(50);
84.   BEGIN
85.     update_employee_manager(
86.       employee_id_in        => l_employee_id,
87.       new_manager_id_in     => l_new_manager_id,
88.       employee_name_out     => l_employee_name,
89.       old_manager_name_out  => l_old_manager_name,
```

Evaluation Copy

```
90.       new_manager_name_out  => l_new_manager_name
91.     );
92.     DBMS_OUTPUT.PUT_LINE('Employee:    ' || l_employee_name);
93.     DBMS_OUTPUT.PUT_LINE('Old Manager: ' || l_old_manager_name);
94.     DBMS_OUTPUT.PUT_LINE('New Manager: ' || l_new_manager_name);
95.   EXCEPTION
96.     WHEN OTHERS THEN
97.       RAISE_APPLICATION_ERROR(-20001, 'An error was encountered - ' ||
98.                                       SQLCODE ||
99.                                       ' -ERROR- ' ||
100.                                      SQLERRM, TRUE);
101.  END;
```

4.7. Naming Unnamed Predefined Exceptions

Earlier in the lesson, we showed a table of the predefined exceptions for which Oracle provides names. There are hundreds of additional predefined exceptions that are not named, but PL/SQL provides a way to assign a name to an exception using `PRAGMA EXCEPTION_INIT`. The syntax is as follows:

Evaluation Copy

```
exception_name EXCEPTION;
PRAGMA EXCEPTION_INIT(exception_name, error_code);
```

This will assign `exception_name` to the predefined exception identified by `error_code`.

For example, the error code `ORA-01400` is caused by an attempt to insert `NULL` into a field that does not allow `NULL` values. To give that error code a name, you would do the following:

```
cannot_insert_null EXCEPTION;
PRAGMA EXCEPTION_INIT(cannot_insert_null, -1400);
```

Notice that 'ORA' is stripped from the error code and so are any leading zeroes. So, `ORA-01400` becomes `-1400`. Likewise, the error code for trying to insert a duplicate value into a field with a unique constraint is `ORA-02291`, which is caused by assigning a foreign key value that doesn't exist in the parent table, would have an error code of `-2291`.

Consider the following code:

EVALUATION COPY: Not to be used in class.

Demo 4.7: Exceptions/Demos/cannot-insert-null.sql

```
1.   BEGIN
2.     INSERT INTO departments
3.     (department_name, manager_id, location_id)
4.     VALUES ('Social Media', 201, 1200);
5.   END;
```

Code Explanation

Run the code. You should get the following error:

```
Error report -
ORA-01400: cannot insert NULL into ("HR"."DEPARTMENTS"."DEPARTMENT_ID")
ORA-06512: at line 2
01400. 00000 -  "cannot insert NULL into (%s)"
*Cause:    An attempt was made to insert NULL into previously listed objects.
*Action:   These objects cannot accept NULL values.
```

Evaluation Copy

This error tells you two important things:

1. The cause:

   ```
   cannot insert NULL into ("HR"."DEPARTMENTS"."DEPARTMENT_ID")
   ```

2. The error code: `ORA-01400`

Armed with this knowledge, you can catch the exception and output a friendlier error message:

Demo 4.8: Exceptions/Demos/cannot-insert-null-caught.sql

```
1.   DECLARE
2.     cannot_insert_null EXCEPTION;
3.     PRAGMA EXCEPTION_INIT(cannot_insert_null, -1400);
4.   BEGIN
5.     INSERT INTO departments
6.     (department_name, manager_id, location_id)
7.     VALUES ('Social Media', 201, 1200);
8.   EXCEPTION
9.     WHEN cannot_insert_null THEN
10.      DBMS_OUTPUT.PUT_LINE('ERROR: Cannot insert NULL.');
11.  END;
```

Code Explanation

Evaluation Copy

Note that our error report, while simpler, is also less informative. We no longer know which field is causing the problem. As such, you may prefer not to catch this exception and let Oracle report it in the default manner. It largely depends who your audience is. If it's another developer, you want the error to be as informative as possible. If it's an end user, you might wish to give them something more friendly.

See the Oracle documentation[4] for more information on unnamed exceptions.

4.8. WHILE Loops

We have used the following code to check for reciprocal employee-manager relationships:

4. https://docs.oracle.com/en/database/oracle/oracle-database/19/errmg/ORA-00000.html

EVALUATION COPY: Not to be used in class.

```
-- Find new manager's manager_id and make sure employees
-- are not reporting to each other
SELECT manager_id
INTO l_managers_manager_id
FROM employees
WHERE employee_id = new_manager_id_in;

IF employee_id_in = l_managers_manager_id THEN
  RAISE reciprocal_managers;
END IF;
```

Again, this checks to make sure that employee id `102` isn't reporting to employee id `103` at the same time that employee id `103` is reporting to employee id `102`. This is good, but not perfect. We really should be checking that an employee's manager is not their report or any of their reports' reports. This can be done using the following checks:

1. Does my manager report to me?
2. Does my manager's manager report to me?
3. Does my manager's manager's manager report me?
4. And so on until we reach the head honcho, who reports to nobody.

Evaluation Copy

In PL/SQL, we can accomplish this with a `WHILE` loop. The syntax for a `WHILE` loop is:

```
WHILE conditions LOOP
  -- executable statements
END LOOP;
```

To avoid an infinite loop, one of the executional statements must do something that can potentially change the result of the conditions.

In our case, we will check each manager's manager up through the chain to make sure than none of them reports to the passed-in employee. Once we get to an employee whose `manager_id` is `NULL`, the loop will end. Our `WHILE` loop will look like this:

```
WHILE l_managers_manager_id IS NOT NULL LOOP

  SELECT manager_id
  INTO l_managers_manager_id
  FROM employees
  WHERE employee_id = l_temp_employee_id;

  DBMS_OUTPUT.PUT_LINE(l_temp_employee_id || ' reports to ' ||
                       l_managers_manager_id);

  IF l_managers_manager_id = employee_id_in THEN
    RAISE reciprocal_managers;
  END IF;

  -- Assign manager's manager id to l_temp_employee_id, so that
  -- the next time through the loop, we will compare their
  -- manager's manager to the passed-in employee.
  l_temp_employee_id := l_managers_manager_id;

END LOOP;
```

Evaluation Copy

The following demo shows the `update_employee_manager()` procedure updated to include this check:

EVALUATION COPY: Not to be used in class.

Demo 4.9: Exceptions/Demos/update-employee-manager-4.sql

```
1.   -- Create the procedure
2.   CREATE OR REPLACE PROCEDURE update_employee_manager(
3.     employee_id_in        IN    employees.employee_id%TYPE,
4.     new_manager_id_in     IN    employees.manager_id%TYPE,
5.     employee_name_out     OUT   VARCHAR2,
6.     old_manager_name_out  OUT   VARCHAR2,
7.     new_manager_name_out  OUT   VARCHAR2
8.   ) IS
9.     l_old_manager_id               employees.manager_id%TYPE := 0;
10.    l_new_managers_manager_id      employees.manager_id%TYPE := 0;
11.    l_num_head_honchos             NUMBER(6,0);
12.    manager_is_self                EXCEPTION;
13.    cannot_remove_last_head_honcho EXCEPTION;
14.    reciprocal_managers            EXCEPTION;
15.
16.    -- Default l_temp_employee_id to new_manager_id_in. Used to check
17.    -- that employee_id_in doesn't report to l_temp_employee_id's boss
18.    l_temp_employee_id employees.employee_id%TYPE := new_manager_id_in;
19.  BEGIN
20.    IF employee_id_in = new_manager_id_in THEN
21.      RAISE manager_is_self;
22.    END IF;
23.
24.    -- Get manager id and name of employee
25.    SELECT manager_id, first_name || ' ' || last_name
26.    INTO l_old_manager_id, employee_name_out
27.    FROM employees
28.    WHERE employee_id = employee_id_in;
29.
30.    -- Get name of old manager
31.    IF l_old_manager_id IS NOT NULL THEN
32.      SELECT first_name || ' ' || last_name
33.      INTO old_manager_name_out
34.      FROM employees
35.      WHERE employee_id = l_old_manager_id;
36.    END IF;
37.
38.    IF new_manager_id_in IS NOT NULL THEN
39.      -- Get full name of the passed-in manager
40.      SELECT first_name || ' ' || last_name
41.      INTO new_manager_name_out
42.      FROM employees
43.      WHERE employee_id = new_manager_id_in;
44.
```

Evaluation Copy

```
45.       -- Make sure employee is not reporting to an underling
46.       WHILE l_new_managers_manager_id IS NOT NULL LOOP
47.         SELECT manager_id
48.         INTO l_new_managers_manager_id
49.         FROM employees
50.         WHERE employee_id = l_temp_employee_id;
51.
52.         DBMS_OUTPUT.PUT_LINE(l_temp_employee_id || ' reports to ' ||
53.                              l_new_managers_manager_id);
54.
55.         IF l_new_managers_manager_id = employee_id_in THEN
56.           RAISE reciprocal_managers;
57.         END IF;
58.         -- Assign manager's manager id to l_temp_employee_id
59.         l_temp_employee_id := l_new_managers_manager_id;
60.
61.       END LOOP;
62.
63.       -- Get number of head honchos not including this employee
64.       -- This will be the number of head honchos left after assigning
65.       -- this employee a manager.
66.       SELECT COUNT(employee_id)
67.       INTO l_num_head_honchos
68.       FROM employees
69.       WHERE manager_id IS NULL AND employee_id <> employee_id_in;
70.
71.       -- Cannot remove last head honcho
72.       IF l_num_head_honchos = 0 THEN
73.         RAISE cannot_remove_last_head_honcho;
74.       END IF;
75.     END IF;
76.
77.     -- Set new manager
78.     UPDATE employees
79.     SET manager_id = new_manager_id_in
80.     WHERE employee_id = employee_id_in;
81.
82.   EXCEPTION
83.     WHEN manager_is_self THEN
84.       RAISE_APPLICATION_ERROR(-20101,
85.                               'Employee cannot be own manager.');
86.     WHEN reciprocal_managers THEN
87.       RAISE_APPLICATION_ERROR(-20102,
88.                           'Employee cannot be managed by an underling.');
89.     WHEN cannot_remove_last_head_honcho THEN
```

Evaluation Copy

EVALUATION COPY: Not to be used in class.

```
90.       RAISE_APPLICATION_ERROR(-20103,
91.                               'Cannot remove last head honcho.');
92.   END update_employee_manager;
93.
94.   -- Call the procedure
95.   DECLARE
96.     l_employee_id       employees.employee_id%TYPE  :=  102;
97.     l_new_manager_id    employees.manager_id%TYPE   :=  107;
98.     l_employee_name     VARCHAR2(50);
99.     l_old_manager_name  VARCHAR2(50);
100.    l_new_manager_name  VARCHAR2(50);
101.  BEGIN
102.    update_employee_manager(
103.      employee_id_in        => l_employee_id,
104.      new_manager_id_in     => l_new_manager_id,
105.      employee_name_out     => l_employee_name,
106.      old_manager_name_out  => l_old_manager_name,
107.      new_manager_name_out  => l_new_manager_name
108.    );
109.    DBMS_OUTPUT.PUT_LINE('Employee:    ' || l_employee_name);
110.    DBMS_OUTPUT.PUT_LINE('Old Manager: ' || l_old_manager_name);
111.    DBMS_OUTPUT.PUT_LINE('New Manager: ' || l_new_manager_name);
112.  EXCEPTION
113.    WHEN OTHERS THEN
114.      RAISE_APPLICATION_ERROR(-20001, 'An error was encountered - ' ||
115.                                      SQLCODE ||
116.                                      ' -ERROR- ' ||
117.                                      SQLERRM, TRUE);
118.  END;
```

Evaluation Copy

Code Explanation

Run the procedure code to create the procedure and then run the block of code below it. You will get a long error message. Above that message, you should see:

```
107 reports to 103
103 reports to 102
```

That is generated from the output in the `WHILE` loop. We are trying to make employee `102` report to employee `107`. But employee `107` reports to employee `103` who in turn reports to `102`. So, if the update were successful, we would end up with:

1. 107 reports to 103.

2. 103 reports to 102.
3. 102 reports to 107.

Our new exception handling prevents this.

4.9. When to Use Exceptions

You are not required to handle all exceptions. In some cases, it may make more sense to leave out the `EXCEPTION` part of the block and let exceptions occur and get reported by Oracle in the default way. Consider the PL/SQL block we used to call the `update_employee_manager()` procedure:

```
DECLARE
  l_employee_id     employees.employee_id%TYPE  :=  101;
  l_new_manager_id  employees.manager_id%TYPE   :=  101;
  l_employee_name   VARCHAR2(50);
  l_old_manager     VARCHAR2(50);
  l_new_manager     VARCHAR2(50);
BEGIN
  update_employee_manager(
    employee_id_in      => l_employee_id,
    new_manager_id_in       => l_new_manager_id,
    employee_name_out   => l_employee_name,
    old_manager_out     => l_old_manager,
    new_manager_out     => l_new_manager
  );
  DBMS_OUTPUT.PUT_LINE('Employee:    ' || l_employee_name);
  DBMS_OUTPUT.PUT_LINE('Old Manager: ' || l_old_manager);
  DBMS_OUTPUT.PUT_LINE('New Manager: ' || l_new_manager);
EXCEPTION
  WHEN OTHERS THEN
    RAISE_APPLICATION_ERROR(-20001, 'An error was encountered - ' ||
                                    SQLCODE ||
                                    ' -ERROR- ' ||
                                    SQLERRM, TRUE);
END;
```

Evaluation Copy

When we run this passing the same value for `employee_id_in` and `new_manager_id_in`, we get the following output:

EVALUATION COPY: Not to be used in class.

```
Error report -
ORA-20001: An error was encountered - -20101 -ERROR- ORA-20101: Employee cannot be own
manager.
ORA-06512: at line 20
ORA-20101: Employee cannot be own manager.
ORA-06512: at "HR.UPDATE_EMPLOYEE_MANAGER", line 54
ORA-06512: at line 8
```

Running the same block after removing the `EXCEPTION` part will give us this output:

```
Error report -
ORA-20101: Employee cannot be own manager.
ORA-06512: at "HR.UPDATE_EMPLOYEE_MANAGER", line 54
ORA-06512: at line 8
```

Evaluation Copy

In this case, we just want the output to be informative so that we can debug the problem. Both reports give us the same important information:

```
ORA-20101: Employee cannot be own manager.
ORA-06512: at "HR.UPDATE_EMPLOYEE_MANAGER", line 54
```

So the `EXCEPTION` part of this block doesn't really help us.

Our recommendation is to let Oracle report exceptions in the default way unless you have some specific reason (e.g., providing a friendlier error message to an end user or suppressing the exception altogether) for catching and handling them.

Conclusion

In this lesson, you have learned to catch and handle exceptions in PL/SQL.

EVALUATION COPY: Not to be used in class.

LESSON 5
Cursors

Topics Covered

☑ What is a cursor?

☑ Implicit cursors.

☑ Explicit cursors.

Introduction

Cursors are pointers to stored information about `SELECT`, `INSERT`, `UPDATE`, and `DELETE` statements.

There are two types of cursors in PL/SQL:

1. **Implicit cursors** - cursors created automatically every time a `SELECT` or DML statement is run.
2. **Explicit cursors** - cursors created and named by the developer.

Evaluation Copy

We will cover both types in this lesson.

5.1. Implicit Cursors

An implicit cursor is created automatically every time a `SELECT` or DML (`INSERT`, `UPDATE`, and `DELETE`) statement is run. When an implicit cursor is created, it is automatically assigned several attributes, the most useful of which are:

1. `SQL%FOUND` - A Boolean value:
    - `TRUE` if at least one row was returned or affected.
    - `FALSE` if no rows were returned or affected.
    - `NULL` if no statement has been run.
2. `SQL%NOTFOUND` - A Boolean value:

- `FALSE` if at least one row was returned or affected.
- `TRUE` if no rows were returned or affected.
- `NULL` if no statement has been run.

3. `SQL%ROWCOUNT` - An integer indicating the number of rows returned or affected.

These attributes always contain information on the last `SELECT` or DML statement that ran, so if you want to maintain information about a particular statement, you should assign the attributes to variables immediately after the statement runs.

❖ 5.1.1. SQL%FOUND

The following example shows how to use `SQL%FOUND` to check if an `UPDATE` statement affected any rows:

Demo 5.1: Cursors/Demos/update-employee-email.sql

```
1.    -- Create procedure
2.    CREATE OR REPLACE PROCEDURE update_employee_email(
3.      employee_id_in  IN  employees.employee_id%TYPE,
4.      new_email_in    IN  employees.email%TYPE
5.    ) IS
6.      no_such_employee EXCEPTION;
7.    BEGIN
8.
9.      UPDATE employees
10.     SET email = new_email_in
11.     WHERE employee_id = employee_id_in;
12.
13.     IF SQL%NOTFOUND THEN
14.       RAISE no_such_employee;
15.     END IF;
16.
17.   EXCEPTION
18.     WHEN no_such_employee THEN
19.       RAISE_APPLICATION_ERROR(-20101,
20.                               'No employee with id ' || employee_id_in);
21.
22.   END update_employee_email;
      -------Lines 23 through 39 Omitted-------
```

Evaluation Copy

EVALUATION COPY: Not to be used in class.

Code Explanation

In the procedure, after updating the email in the `employees` table, we use `SQL%NOTFOUND` to check if any rows were affected. If no rows were affected, we raise a `no_such_employee` exception.

1. Run the code to create the `update_employee_email()` procedure.
2. Run the block of code below the procedure to call it. Notice we're passing in `51` for the `employee_id`. There is no employee with that id. This will result in the following output:

```
ORA-20001: An error was encountered - -20101 -ERROR- ORA-20101: No employee with
 id 51
```

Evaluation Copy

❖ 5.1.2. SQL%ROWCOUNT

The following example shows how to use `SQL%ROWCOUNT` to see how many rows are affected by a `DELETE` statement:

Demo 5.2: Cursors/Demos/drop_department.sql

```
1.   -- Create procedure
2.   CREATE OR REPLACE PROCEDURE drop_department(
3.     department_id_in  IN    departments.department_id%TYPE,
4.     num_layoffs_out   OUT   NUMBER
5.   ) IS
6.     no_such_department EXCEPTION;
7.   BEGIN
8.
9.     UPDATE departments
10.    SET manager_id = NULL
11.    WHERE department_id = department_id_in;
12.
13.    IF SQL%NOTFOUND THEN
14.      RAISE no_such_department;
15.    END IF;
16.
17.    DELETE FROM employees
18.    WHERE department_id = department_id_in;
19.
20.    num_layoffs_out := SQL%ROWCOUNT;
21.
22.    DELETE FROM departments
23.    WHERE department_id = department_id_in;
24.
25.  EXCEPTION
26.    WHEN no_such_department THEN
27.      RAISE_APPLICATION_ERROR(-20101,
28.                      'No department with id ' || department_id_in);
29.
30.  END drop_department;
31.
32.  -- Call procedure
33.  DECLARE
34.    l_department_id departments.department_id%TYPE := 100;
35.    l_num_layoffs   NUMBER(6,0);
36.  BEGIN
37.    drop_department(
38.      department_id_in  => l_department_id,
39.      num_layoffs_out   => l_num_layoffs
40.    );
41.    DBMS_OUTPUT.PUT_LINE(l_num_layoffs || ' employees laid off.');
42.  EXCEPTION
43.    WHEN OTHERS THEN
44.      RAISE_APPLICATION_ERROR(-20001, 'An error was encountered - ' ||
```

Evaluation Copy

EVALUATION COPY: Not to be used in class.

```
45.                                     SQLCODE ||
46.                                     ' -ERROR- ' ||
47.                                     SQLERRM, TRUE);
48.   END;
```

Code Explanation

Things to notice:

1. In the procedure, we first update the `departments` table to remove the `manager_id`. We need to do this before deleting this department's employees from the `employees` table. Otherwise, we would get this error:

   ```
   ORA-02292: integrity constraint (HR.DEPT_MGR_FK) violated - child record found
   ```

   This is because one of the employees we're attempting to delete is the manager in the `departments` table.

2. If the update doesn't affect any rows, that means the `department_id` doesn't exist, in which case we raise a `no_such_department` exception.

3. After removing the manager from the department, we delete all employees from the `employees` table that were in the department matching the passed-in `department_id`.

4. We then assign `SQL%ROWCOUNT` to save the number of affected rows (deleted records) to `num_layoffs_out`.

Evaluation Copy

In the block below the procedure, we call `drop_department()`, passing in `100` for `department_id_in`. The resulting output will be:

```
6 employees laid off.
```

Try passing `51` for `department_id_in` and re-running the block. You should get an exception:

```
ORA-20001: An error was encountered - -20101 -ERROR- ORA-20101: No department with id
51
```

Execute a `ROLLBACK` to undo these changes:

```
ROLLBACK;
```

Exercise 11: Using Implicit Cursor Attributes

10 to 15 minutes

Open `Cursors/Exercises/merge-departments.sql` in SQL Developer and review the code.

1. Review the code in the `merge_departments()` procedure carefully. You should be able to understand how it works.
2. Beneath the procedure, we make a call to it passing in `100` for `department_id_keep_in` and `110` for `department_id_lose_in`.

Your job is to use cursor attributes within the procedure:

1. Below the first `UPDATE` statement, add code to assign the number of affected rows to `employees_merged_out`.
2. Instead of deleting the department that is going away, we just update it to remove the manager. Below this second `UPDATE` statement, raise a `no_such_department_to_lose` exception if no department was updated.
3. Run the code to compile the procedure. Fix any errors you get until it compiles without errors.
4. Run the block below the procedure. It should output:

```
2 employees merged into department 100
```

5. Now, in the block below the procedure, change the values of:
   A. `department_id_keep` to `105`, which doesn't exist.
   B. `department_id_lose` to `100`, which does exist.

   You should get a error with the message:

```
Cannot keep department 105. Does not exist.
```

6. Now change the values of:
   A. `department_id_keep` to `100`, which does exist.
   B. `department_id_lose` to `105`, which doesn't exist.

Evaluation Copy

EVALUATION COPY: Not to be used in class.

You should get a error with the message:

```
Cannot lose department 105. Does not exist.
```

7. Execute a `ROLLBACK` to undo these changes:

```
ROLLBACK;
```

Evaluation Copy

Solution: Cursors/Solutions/merge-departments.sql

```
1.   -- Create procedure
2.   CREATE OR REPLACE PROCEDURE merge_departments(
3.     department_id_keep_in  IN   departments.department_id%TYPE,
4.     department_id_lose_in  IN   departments.department_id%TYPE,
5.     employees_merged_out   OUT  NUMBER
6.   ) IS
7.     integrity_constraint_violated EXCEPTION;
8.     PRAGMA EXCEPTION_INIT(integrity_constraint_violated, -2291);
9.     no_such_department_to_lose EXCEPTION;
10.  BEGIN
11.
12.    -- Will cause integrity_constraint_violated error if
13.    -- no department_id is department_id_keep_in
14.    UPDATE employees
15.    SET department_id = department_id_keep_in
16.    WHERE department_id = department_id_lose_in;
17.
18.    employees_merged_out := SQL%ROWCOUNT;
19.
20.    UPDATE departments
21.    SET manager_id = NULL
22.    WHERE department_id = department_id_lose_in;
23.
24.    IF SQL%NOTFOUND THEN
25.      RAISE no_such_department_to_lose;
26.    END IF;
27.
28.  EXCEPTION
29.    WHEN integrity_constraint_violated THEN
30.      RAISE_APPLICATION_ERROR(-20101, 'Cannot keep department ' ||
31.                    department_id_keep_in || '. Does not exist.');
32.    WHEN no_such_department_to_lose THEN
33.      RAISE_APPLICATION_ERROR(-20102, 'Cannot lose department ' ||
34.                    department_id_lose_in || '. Does not exist.');
35.
36.  END merge_departments;
     -------Lines 37 through 58 Omitted-------
```

Evaluation Copy

EVALUATION COPY: Not to be used in class.

5.2. Explicit Cursors

Explicit cursors are cursors that the developer declares and defines. They are used to handle records returned by a `SELECT` statement.

The steps involved in working with explicit cursors are as follows:

1. Declare and define the cursor:

```
CURSOR cursor_name IS
  SELECT column_names
  -- rest of query
```

   For example:

```
CURSOR employee_cursor IS
  SELECT first_name, last_name, salary
  FROM employees;
```

2. Open the cursor:

Evaluation Copy

```
OPEN cursor_name;
```

   For example:

```
OPEN employee_cursor;
```

3. Fetch a row from the cursor:

```
FETCH cursor_name INTO variable(s)
```

   For example:

```
FETCH employee_cursor INTO l_first_name, l_last_name, l_salary;
```

   Note that `l_first_name`, `l_last_name`, and `l_salary` must be defined before the fetch and their data types must be compatible with the fields returned by the fetch.

4. Close the cursor:

```
CLOSE cursor_name;
```

For example:

```
CLOSE employee_cursor;
```

Often, cursors are fetched one row at a time within a loop, using `LOOP` to start the loop and `END LOOP` to end it. The syntax for that is shown below:

```
LOOP
  FETCH cursor_name INTO variable(s);
  EXIT WHEN cursor_name%NOTFOUND;
END LOOP;
```

Note the `EXIT WHEN` line. Trying to fetch a record from a cursor after having already fetched the last record will not result in an exception. You must explicitly exit the loop when you come to the end of the cursor. As demonstrated above, you can use the `%NOTFOUND` attribute to check this.

Putting this all together, the syntax for declaring, defining, opening, looping through and fetching, and closing a cursor is shown below in the context of a PL/SQL block:

Evaluation Copy

```
DECLARE
  CURSOR cursor_name
    SELECT column_names
    -- rest of query

  -- variable declarations
BEGIN
  OPEN cursor_name;

  LOOP
    FETCH cursor_name INTO variable(s);
    EXIT WHEN cursor_name%NOTFOUND;
  END LOOP;

  CLOSE cursor_name;
END;
```

And here is a simple working example:

EVALUATION COPY: Not to be used in class.

Demo 5.3: Cursors/Demos/explicit-cursor-simple.sql

```
1.   DECLARE
2.     CURSOR employee_cursor IS
3.       SELECT first_name || ' ' || last_name AS full_name, salary
4.       FROM employees;
5.
6.     l_full_name  VARCHAR2(50);
7.     l_salary     employees.salary%TYPE;
8.
9.   BEGIN
10.    OPEN employee_cursor;
11.
12.    LOOP
13.      FETCH employee_cursor INTO l_full_name, l_salary;
14.      EXIT WHEN employee_cursor%NOTFOUND;
15.      DBMS_OUTPUT.PUT_LINE(
16.        RPAD(l_full_name, 50) || ': ' || to_char(l_salary, '$99,999')
17.      );
18.    END LOOP;
19.
20.    CLOSE employee_cursor;
21.  END;
```

Evaluation Copy

Code Explanation

Run this file in SQL Developer. It should output the following (first 10 lines shown):

```
Steven King                                        :  $24,000
Neena Kochhar                                      :  $17,000
Lex De Haan                                        :  $17,000
Alexander Hunold                                   :   $9,000
Bruce Ernst                                        :   $6,000
David Austin                                       :   $4,800
Valli Pataballa                                    :   $4,800
Diana Lorentz                                      :   $4,200
Nancy Greenberg                                    :  $12,008
Daniel Faviet                                      :   $9,000
```

5.3. %ROWTYPE

You have already seen that you can declare a variable to be of the same type as a field from a table in the database using `%TYPE`. You can also declare a variable with `%ROWTYPE` that represents a complete record (row) from a table:

```
l_variable_name cursor_name%ROWTYPE;
```

For example:

```
l_employee employee_cursor%ROWTYPE;
```

In this case, you reference the individual fields using dot notation as shown in the example below:

Demo 5.4: Cursors/Demos/explicit-cursor-rowtype.sql

```
1.   DECLARE
2.     CURSOR employee_cursor IS
3.       SELECT first_name || ' ' || last_name AS full_name, salary
4.       FROM employees;
5.
6.     l_employee employee_cursor%ROWTYPE;
7.
8.   BEGIN
9.     OPEN employee_cursor;
10.
11.    LOOP
12.      FETCH employee_cursor INTO l_employee;
13.      EXIT WHEN employee_cursor%NOTFOUND;
14.      DBMS_OUTPUT.PUT_LINE(
15.        RPAD(l_employee.full_name, 50) || ': ' ||
16.        to_char(l_employee.salary, '$99,999')
17.      );
18.    END LOOP;
19.
20.    CLOSE employee_cursor;
21.  END;
```

Evaluation Copy

EVALUATION COPY: Not to be used in class.

Generally, when using explicit cursors, you will do more than just output data from the `SELECT` statement. You might, for example, pass data record by record to a PL/SQL subprogram to make modifications in the database. Next, we'll take a look at a more complete example that does just that.

5.4. Explicit Cursor Use Case

Evaluation Copy

To help you understand when you would use an explicit cursor, let's consider this example. Suppose we're going through hard times and we have to reduce the monthly payroll by $30,000. We decide that instead of layoffs, we are going to reduce salaries. Starting with the latest hire, we will reduce each employee's salary by 5% until we have reduced total payroll by $30,000. We'll use the following procedure to reduce an employee's salary:

Demo 5.5: Cursors/Demos/proc-change-salary.sql

```
     -------Line 1 Omitted-------
2.   CREATE OR REPLACE PROCEDURE change_salary(
3.     employee_id_in  IN    employees.employee_id%TYPE,
4.     change_amount   IN    NUMBER, -- decimal representing % change
5.     salary_old_out  OUT   employees.salary%TYPE,
6.     salary_new_out  OUT   employees.salary%TYPE
7.   ) IS
8.     no_such_employee EXCEPTION;
9.   BEGIN
10.    SELECT salary
11.    INTO salary_old_out
12.    FROM employees
13.    WHERE employee_id = employee_id_in;
14.
15.    UPDATE employees
16.    SET salary = salary * (1 + change_amount)
17.    WHERE employee_id = employee_id_in
18.    RETURNING salary INTO salary_new_out;
19.
20.    IF SQL%NOTFOUND THEN
21.      RAISE no_such_employee;
22.    END IF;
23.
24.  EXCEPTION
25.    WHEN no_such_employee THEN
26.      RAISE_APPLICATION_ERROR(-20101,
27.                              'No employee with id ' || employee_id_in);
28.
29.  END change_salary;
```

Evaluation Copy

Code Explanation

Study to procedure. It shouldn't be difficult to understand. Note that it has two `out` parameters to store the salary before and after the change:

1. `salary_old_out`
2. `salary_new_out`

Run the code to create the procedure.

EVALUATION COPY: Not to be used in class.

We will create a cursor and loop through it, calling the `change_salary()` procedure repeatedly, passing in employee ids until we have reached our goal of a $30,000 total salary reduction. Here's the code to do that:

Demo 5.6: Cursors/Demos/reduce-payroll.sql

```
1.   DECLARE
2.      l_amt_to_reduce  NUMBER(10,2) := 30000;
3.      l_change_amt     NUMBER(2,2)  := -0.05;
4.      l_i              NUMBER(3,0)  :=1;
5.
6.      CURSOR employee_cursor
7.      IS
8.         SELECT employee_id, first_name, last_name
9.         FROM employees
10.        ORDER BY hire_date DESC;
11.
12.     l_employee   employee_cursor%ROWTYPE;
13.     l_salary_old employees.salary%TYPE;
14.     l_salary_new employees.salary%TYPE;
15.  BEGIN
16.     OPEN employee_cursor;
17.
18.     LOOP
19.        FETCH employee_cursor INTO l_employee;
20.        EXIT WHEN employee_cursor%NOTFOUND;
21.
22.        change_salary(l_employee.employee_id, l_change_amt,
23.                              l_salary_old, l_salary_new);
24.        l_amt_to_reduce := l_amt_to_reduce -
25.                              (l_salary_old - l_salary_new);
26.
27.        DBMS_OUTPUT.PUT_LINE(
28.          l_i || '. Salary of ' || l_employee.first_name || ' ' ||
29.          l_employee.last_name || ' reduced from ' ||
30.          to_char(l_salary_old, '$99,999') || ' to ' ||
31.          to_char(l_salary_new, '$99,999') || '. Left to reduce: ' ||
32.          to_char(l_amt_to_reduce, '$99,999') || '.'
33.        );
34.        l_i := l_i + 1;
35.        EXIT WHEN l_amt_to_reduce <= 0;
36.     END LOOP;
37.
38.     CLOSE employee_cursor;
39.  END;
```

Evaluation Copy

Code Explanation

Run the code. It should output the following (first and last 5 lines shown):

```
1. Salary of Sundita Kumar reduced from   $6,100 to   $5,795. Left to reduce:  $29,695.
2. Salary of Amit Banda reduced from   $6,200 to   $5,890. Left to reduce:  $29,385.
3. Salary of Sundar Ande reduced from   $6,400 to   $6,080. Left to reduce:  $29,065.
4. Salary of Steven Markle reduced from   $2,200 to   $2,090. Left to reduce:  $28,955.
5. Salary of David Lee reduced from   $6,800 to   $6,460. Left to reduce:  $28,615.
...
95. Salary of Jennifer Whalen reduced from   $4,400 to   $4,180. Left to reduce:   $1,650.
96. Salary of Renske Ladwig reduced from   $3,600 to   $3,420. Left to reduce:   $1,470.
97. Salary of Steven King reduced from  $24,000 to  $22,800. Left to reduce:     $270.
98. Salary of Alexander Khoo reduced from   $3,100 to   $2,945. Left to reduce:     $115.
99. Salary of Payam Kaufling reduced from   $7,900 to   $7,505. Left to reduce:    -$280.
```

Note the following:

1. The cursor is declared and defined as follows:

   ```
   CURSOR employee_cursor  IS
     SELECT employee_id, first_name, last_name
     FROM employees
     ORDER BY hire_date DESC;
   ```

   This creates a cursor from which we can fetch records one by one.

2. Within the loop, we fetch each record into the `l_employee` record, which was defined as `employee_cursor%ROWTYPE`:

   ```
   FETCH employee_cursor INTO l_employee;
   ```

3. After each fetch, and before calling the `change_salary()` procedure, we check if a record was actually found and we exit the loop if none was:

   ```
   EXIT WHEN employee_cursor%NOTFOUND;
   ```

Execute a `ROLLBACK` to undo these changes:

Evaluation Copy

EVALUATION COPY: Not to be used in class.

```
ROLLBACK;
```

5.5. Cursor FOR LOOP

A cursor `FOR LOOP` makes looping through a cursor a little easier. It handles the opening, iterative fetching, and closing of the cursor. The syntax is as follows:

```
FOR item IN cursor_name LOOP
  /*
    Code to execute including optional EXIT WHEN statement
  */
END LOOP;
```

Here is the previous example modified to use a cursor `FOR LOOP`:

Demo 5.7: Cursors/Demos/reduce-payroll-cursor-for-loop.sql

Evaluation Copy

```
     -------Lines 1 through 12 omitted -------
13.  BEGIN
14.
15.     FOR l_employee IN employee_cursor LOOP
16.        change_salary(l_employee.employee_id, l_change_amt,
17.                              l_salary_old, l_salary_new);
18.        l_amt_to_reduce := l_amt_to_reduce -
19.                              (l_salary_old - l_salary_new);
20.
21.        DBMS_OUTPUT.PUT_LINE(
22.          l_i || '. Salary of ' || l_employee.first_name || ' ' ||
23.          l_employee.last_name || ' reduced from ' ||
24.          to_char(l_salary_old, '$99,999') || ' to ' ||
25.          to_char(l_salary_new, '$99,999') || '. Left to reduce: ' ||
26.          to_char(l_amt_to_reduce, '$99,999') || '.'
27.        );
28.        l_i := l_i + 1;
29.        EXIT WHEN l_amt_to_reduce <= 0;
30.     END LOOP;
31.
32.  END;
```

Exercise 12: Using an Explicit Cursor

45 to 90 minutes

In this exercise, you will use explicit cursors to restructure the data in our database. Note that there is a lot going on in this exercise. You will not only get good practice with explicit cursors, but will review a lot of the PL/SQL concepts you have learned so far, and you will also get some insight into database design.

❖ E12.1. Current employees Table

Currently, there is an `employees` table with the following fields:

1. `employee_id`
2. `first_name`
3. `last_name`
4. `email`
5. `phone_number`
6. `hire_date`
7. `job_id`
8. `salary`
9. `commission_pct`
10. `manager_id`
11. `department_id`

Evaluation Copy

❖ E12.2. New clients and vendors Tables

Run the script at `Cursors/Exercises/create-clients-and-vendors.sql` to create and populate `clients` and `vendors` tables. If you already have `clients` and `vendors` tables in the `HR` database then you should first drop those tables using the `DROP` statements in the comment at the top of the file.

Both tables have these fields:

1. `client_id` / `vendor_id`

EVALUATION COPY: Not to be used in class.

2. `first_name`
3. `last_name`
4. `email`
5. `phone_number`
6. `company_name`

Note that we now have three tables with the following fields:

1. `first_name`
2. `last_name`
3. `email`
4. `phone_number`

Having the same structure of data in multiple tables can create extra work for us. Imagine you want to create a `get_phone_number()` function. You have to either create separate functions for employees, clients, and vendors, or you have to pass in a parameter to the function so that it will know which table to query. Further imagine that you start collecting data on partners. You will need to create a new `partners` table with these same fields and then either create a new `get_partner_phone_number()` function or modify your `get_phone_number()` function to account for the `partners` table.

Evaluation Copy

❖ E12.3. New Tables

To deal with this problem, we have decided to combine this data into a single `people` table that will have these same fields, plus a `company_id` field that will be a foreign key referencing the primary key of a new `companies` table.

In addition, because the `employees` table contains additional fields, we will create a table called `people_employees` to hold this additional data. It will include a `person_id` foreign key referencing the primary key of the `people` table, so that we can tie this extra data to a specific person.

Companies stored in the `companies` table could be one or more of the following types:

1. `staff` - this means it is our company. There will only be one of these.
2. `client` - for companies that are clients.
3. `vendor` - for companies that are vendors.

The table will include fields for those three types. Ideally, we would want those fields to be Boolean (`TRUE`/`FALSE`) values, but standard SQL doesn't include a `BOOLEAN` type, so we will use `CHAR(1)` and restrict its values to '1' (for `TRUE`) and '0' (for `FALSE`). This structure allows us to mark a company as both a client and a vendor, as it's possible for a company to be both:

COMPANY_ID	COMPANY_NAME	STAFF	VENDOR	CLIENT
12	Visa	0	1	1

The `companies`, `people`, and `people_employees` tables are created in the file below. Review the code and make sure you understand how these tables are designed:

EVALUATION COPY: Not to be used in class.

Exercise Code 12.1: Cursors/Exercises/create-new-tables.sql

```
1.   DROP TABLE people_employees;
2.   DROP TABLE people;
3.   DROP TABLE companies;
4.
5.   /*
6.     In the companies table below, the staff, vendor, and client
7.     columns are pseudo-booleans.
8.       '1' = True, meaning the company falls in this category
9.       '0' = False, meaning the company does not fall in this category
10.  */
11.  CREATE TABLE companies
12.  (
13.    company_id       NUMBER GENERATED AS IDENTITY,
14.    company_name     VARCHAR2(100) UNIQUE,
15.    staff            CHAR(1) DEFAULT '0' NOT NULL,
16.    vendor           CHAR(1) DEFAULT '0' NOT NULL,
17.    client           CHAR(1) DEFAULT '0' NOT NULL,
18.    CONSTRAINT PK_company PRIMARY KEY (company_id),
19.    CONSTRAINT cons_companies_staff  CHECK (staff IN ('1','0')),
20.    CONSTRAINT cons_companies_vendor CHECK (vendor IN ('1','0')),
21.    CONSTRAINT cons_companies_client CHECK (client IN ('1','0'))
22.  );
23.
24.  CREATE TABLE people
25.  (
26.    person_id       NUMBER GENERATED AS IDENTITY,
27.    first_name      VARCHAR2(20),
28.    last_name       VARCHAR2(25)    NOT NULL,
29.    email           VARCHAR2(25)    NOT NULL,
30.    phone_number    VARCHAR2(20),
31.    company_id      NUMBER REFERENCES companies(company_id),
32.    CONSTRAINT PK_people PRIMARY KEY (person_id)
33.  );
34.
35.  CREATE TABLE people_employees
36.  (
37.    employee_id       NUMBER GENERATED AS IDENTITY,
38.    person_id         NUMBER REFERENCES people(person_id),
39.    hire_date         DATE,
40.    job_id            VARCHAR2(10) REFERENCES jobs(job_id),
41.    salary            NUMBER(8,2),
42.    commission_pct    NUMBER(2,2),
43.    manager_id        NUMBER(6,0),
44.    department_id     NUMBER(4) REFERENCES departments(department_id),
```

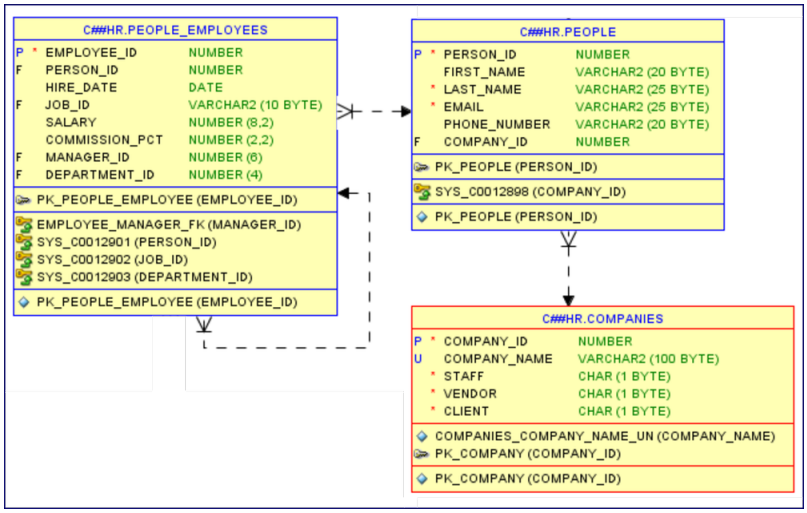
Evaluation Copy

```
45.     CONSTRAINT PK_people_employee PRIMARY KEY (employee_id)
46.   );
47.
48.   -- Create the relationship between manager_id and employee_id
49.   ALTER TABLE people_employees
50.   ADD CONSTRAINT people_employee_manager_fk
51.   FOREIGN KEY (manager_id)
52.   REFERENCES people_employees(employee_id);
```

Code Explanation

Run this file to create the tables.

The structure and relationships of the new table is shown below:



❖ E12.4. The insert_person() Procedure

Below, we have defined a new `insert_person()` procedure based on our new tables. While it would be a good exercise for you to review the whole procedure, the most important thing is that you understand how to call it.

EVALUATION COPY: Not to be used in class.

Exercise Code 12.2: Cursors/Exercises/insert-person.sql

```
1.  CREATE OR REPLACE PROCEDURE insert_person(
2.    first_name_in     IN    people.first_name%TYPE,
3.    last_name_in      IN    people.last_name%TYPE,
4.    email_in          IN    people.email%TYPE,
5.    phone_number_in   IN    people.phone_number%TYPE,
6.    company_name_in   IN    companies.company_name%TYPE,
7.    company_type_in   IN    VARCHAR2, -- staff, client, or vendor
8.    hire_date_in      IN    DATE :=  SYSDATE
9.  ) IS
10.   l_company_id          companies.company_id%TYPE;
11.   l_staff               companies.staff%TYPE  := '0';
12.   l_client              companies.client%TYPE := '0';
13.   l_vendor              companies.vendor%TYPE := '0';
14.   l_staff_person_id     people.person_id%TYPE; -- in case we insert a staff
            person
15.   no_such_company_type  EXCEPTION;
16. BEGIN
17.   -- Save current values into variable if company already in table.
18.   -- If it doesn't (NO_DATA_FOUND), fail silently.
19.   BEGIN
20.     SELECT company_id, staff, client, vendor
21.     INTO l_company_id, l_staff, l_client, l_vendor
22.     FROM companies
23.     WHERE company_name = company_name_in;
24.   EXCEPTION
25.     WHEN NO_DATA_FOUND THEN
26.       NULL;
27.   END;
28.
29.   -- Change values of l_staff, l_client, and l_vendor
30.   -- in preparation for update or insert
31.   CASE company_type_in
32.     WHEN 'staff' THEN
33.       l_staff := '1';
34.     WHEN 'client' THEN
35.       l_client := '1';
36.     WHEN 'vendor' THEN
37.       l_vendor := '1';
38.     ELSE
39.       RAISE no_such_company_type;
40.   END CASE;
41.
42.   -- If l_company_id is NULL, it means this is a new company.
43.   IF l_company_id IS NULL THEN
```

Evaluation Copy

```
44.       INSERT INTO companies
45.       (company_name, staff, client, vendor)
46.       VALUES(company_name_in, l_staff, l_client, l_vendor)
47.       RETURNING company_id INTO l_company_id;
48.     -- Otherwise, a company with this name already existed
49.     -- in the companies table, so we update.
50.     ELSE
51.       UPDATE companies
52.       SET staff   = l_staff,
53.           client  = l_client,
54.           vendor  = l_vendor
55.       WHERE company_name = company_name_in;
56.     END IF;
57.
58.     -- Insert new person into people
59.     INSERT INTO people
60.     (first_name, last_name, email, phone_number, company_id)
61.     VALUES
62.     (first_name_in,last_name_in,email_in,phone_number_in,l_company_id)
63.     RETURNING person_id INTO l_staff_person_id;
64.
65.     -- If new person is staff, insert into people_employees. Really,
66.     -- we'd have IN parameters for all fields in people_employees.
67.     IF company_type_in = 'staff' THEN
68.       INSERT INTO people_employees
69.       (person_id, hire_date)
70.       VALUES (l_staff_person_id, hire_date_in);
71.     END IF;
72.
73.   EXCEPTION
74.     WHEN no_such_company_type THEN
75.       RAISE_APPLICATION_ERROR(-20101,
76.                     'Company type must be staff, client, or vendor.');
77.
78.   END insert_person;
```

Evaluation Copy

Code Explanation

Notice that this file has a nested block starting immediately after the first `BEGIN` statement. This nested block allows us to capture and suppress the exception that will be caused if no company with the passed-in `company_name` is found in the `companies` table. Later on in the code we check if `company_id` is `NULL`. If it is, we insert a new company. If not, we update the existing one.

Run this file to create the procedure.

EVALUATION COPY: Not to be used in class.

You will now use cursors to insert records from the `employees`, `clients` and `vendors` tables into the new tables.

1. Open `Cursors/Exercises/populate-people-tables.sql` in SQL Developer.
2. The employee cursor has already been done. Review that code.
3. Declare cursors for client and vendor. Note that clients and vendors don't have hire dates, but they do have `company_name` fields.
4. Loop through the client and vendor cursors inserting people.
5. When you are finished, run the file and then run the following queries, which you can find in `Cursors/Exercises/populate-people-queries.sql`:

   A. `SELECT COUNT(*) FROM companies;`

      - The result should be `166`.

   B. `SELECT COUNT(*) FROM people;`

      - The result should be `325`.

   C. `SELECT COUNT(*) FROM people_employees;`

      - The result should be `107`.

   D. `SELECT COUNT(*) FROM companies WHERE vendor='1';`

      - The result should be `81`.

   E. `SELECT COUNT(*) FROM companies WHERE client='1';`

      - The result should be `112`.

   F. `SELECT COUNT(*) FROM companies WHERE client='1' and vendor='1';`

      - The result should be `28`.

Evaluation Copy

Solution: Cursors/Solutions/populate-people-tables.sql

```
1.   DECLARE
2.
3.      CURSOR employee_cursor IS
4.         SELECT first_name, last_name, email, phone_number, hire_date
5.         FROM employees;
6.
7.      CURSOR client_cursor IS
8.         SELECT first_name, last_name, email, phone_number, company_name
9.         FROM clients;
10.
11.     CURSOR vendor_cursor IS
12.        SELECT first_name, last_name, email, phone_number, company_name
13.        FROM vendors;
14.
15.  BEGIN
16.
17.    FOR l_employee IN employee_cursor LOOP
18.      insert_person(
19.        first_name_in   => l_employee.first_name,
20.        last_name_in    => l_employee.last_name,
21.        email_in        => l_employee.email,
22.        phone_number_in => l_employee.phone_number,
23.        company_name_in => 'Webucator',
24.        company_type_in => 'staff',
25.        hire_date_in    => l_employee.hire_date
26.      );
27.    END LOOP;
28.
29.    FOR l_client IN client_cursor LOOP
30.      insert_person(
31.        first_name_in   => l_client.first_name,
32.        last_name_in    => l_client.last_name,
33.        email_in        => l_client.email,
34.        phone_number_in => l_client.phone_number,
35.        company_name_in => l_client.company_name,
36.        company_type_in => 'client'
37.      );
38.    END LOOP;
39.
40.    FOR l_vendor IN vendor_cursor LOOP
41.      insert_person(
42.        first_name_in   => l_vendor.first_name,
43.        last_name_in    => l_vendor.last_name,
44.        email_in        => l_vendor.email,
```

Evaluation Copy

EVALUATION COPY: Not to be used in class.

```
45.         phone_number_in => l_vendor.phone_number,
46.         company_name_in => l_vendor.company_name,
47.         company_type_in => 'vendor'
48.       );
49.     END LOOP;
50.
51.   END;
```

5.6. Cursor Parameters

Explicit cursors can take parameters, making them more flexible and thereby more reusable. Here's a simple example:

Demo 5.8: Cursors/Demos/explicit-cursor-parameters.sql

```
1.    DECLARE
2.      CURSOR employee_cursor(min_salary NUMBER) IS
3.        SELECT first_name || ' ' || last_name AS full_name, salary
4.        FROM employees
5.        WHERE salary >= min_salary;
6.
7.    BEGIN
8.
9.      FOR l_employee IN employee_cursor(10000) LOOP
10.       DBMS_OUTPUT.PUT_LINE(
11.         RPAD(l_employee.full_name, 50) || ': ' ||
12.         to_char(l_employee.salary, '$99,999')
13.       );
14.     END LOOP;
15.
16.   END;
```

Evaluation Copy

Code Explanation

Run this file and you'll see that it only outputs employees with salaries greater than or equal to `10000` (first 10 lines shown):

```
Steven King                                :  $24,000
Neena Kochhar                              :  $17,000
Lex De Haan                                :  $17,000
Nancy Greenberg                            :  $12,008
Den Raphaely                               :  $11,000
John Russell                               :  $14,000
Karen Partners                             :  $13,500
Alberto Errazuriz                          :  $12,000
Gerald Cambrault                           :  $11,000
Eleni Zlotkey                              :  $10,500
```

Evaluation Copy

Change the value passed into the cursor to get different results.

Conclusion

In this lesson, you have learned to work with implicit and explicit cursors.

EVALUATION COPY: Not to be used in class.

LESSON 6
Packages

Topics Covered

☑ Packages.

Introduction

In this lesson, you will learn to use packages to group related PL/SQL objects.

6.1. Package Basics

A *package* is a group of related PL/SQL objects:

Evaluation Copy

1. types
2. variables
3. constants
4. subprograms
5. cursors
6. exceptions

❖ 6.1.1. Package Parts

A package is made up of the following parts:

1. **Package Specification** - This is where you declare *public* items that can be used from outside of the package.
2. **Package Body** - This is where the package objects that are declared in the specification are defined. Any objects declared in the package body that are not declared in the specification will be *private*, meaning that they will only be available to other objects within the package.

6.2. The Package Specification

The package specification is used to declare *public* items - items that can be used from outside of the package. The syntax for creating (or replacing) a package is:

```
CREATE OR REPLACE PACKAGE package_name IS

  -- declarations

END package_name;
```

For example, the following code creates a package used to work with circles.

Demo 6.1: Packages/Demos/circle-package.sql

```
1.   CREATE OR REPLACE PACKAGE circle_package
2.   IS
3.
4.    pi CONSTANT FLOAT := 3.1415926535897932;
5.
6.    FUNCTION get_area(radius_in NUMBER) RETURN NUMBER;
7.    FUNCTION get_circumference(radius_in NUMBER) RETURN NUMBER;
8.
9.   END circle_package;
```

Evaluation Copy

Code Explanation

Things to notice:

1. The constant `pi` is declared and defined. You do not need to create the package body to make use of this. After creating this package specification, you could run the following code:

   ```
   BEGIN
     DBMS_OUTPUT.PUT_LINE(circle_package.pi);
   END;
   ```

   And it would output:

   ```
   3.1415926535897932
   ```


EVALUATION COPY: Not to be used in class.

2. The `get_area()` and `get_circumference()` functions are declared, but not defined. They should be defined in the package body, which is required for all packages that have subprograms or cursors.
3. In a package, functions are defined using just the `FUNCTION` keyword. You do not use `CREATE FUNCTION`. Likewise, procedures are defined using just `PROCEDURE`.

6.3. The Package Body

Evaluation Copy

The package body is used to define public package objects (objects that are declared in the package specification), and to declare and define private package objects. The syntax for creating (or replacing) a package body is:

```
CREATE OR REPLACE PACKAGE BODY package_name IS

  -- declarations of public objects
  -- definitions and declarations of private objects

END package_name;
```

For example, the following code creates the package body that goes with our `circle_package` package:

Demo 6.2: Packages/Demos/circle-package-body.sql

```
1.   CREATE OR REPLACE PACKAGE BODY circle_package
2.   IS
3.
4.     -- Private function (not declared in package specification)
5.     FUNCTION get_diameter(radius_in NUMBER)
6.       RETURN NUMBER
7.     IS
8.       -- nothing to declare
9.     BEGIN
10.      RETURN radius_in * 2;
11.    END get_diameter;
12.
13.    -- Public functions (declared in package specification)
14.    FUNCTION get_area(radius_in NUMBER)
15.      RETURN NUMBER
16.    IS
17.      -- nothing to declare
18.    BEGIN
19.      RETURN POWER(radius_in, 2) * pi;
20.    END get_area;
21.
22.    FUNCTION get_circumference(radius_in NUMBER)
23.      RETURN NUMBER
24.    IS
25.      -- nothing to declare
26.    BEGIN
27.      -- calls private function
28.      RETURN get_diameter(radius_in) * pi;
29.    END get_circumference;
30.
31.  END circle_package;
```

Evaluation Copy

Code Explanation

Things to notice:

1. We first declare one private function: `get_diameter()`, which is then used by the public `get_circumference()` function. In practice, the `get_diameter()` function would probably be public too. We make it private here for illustration purposes.
2. We then declare the two public functions defined in the package specification.
3. Note that the `pi` constant defined in the package specification is available in the body.

EVALUATION COPY: Not to be used in class.

❖ 6.3.1. Using the Package

The following example shows how to make use of the package:

Demo 6.3: Packages/Demos/use-circle-package.sql

```
1.   DECLARE
2.     l_area            NUMBER(8,2);
3.     l_circumference   NUMBER(8,2);
4.   BEGIN
5.     DBMS_OUTPUT.PUT_LINE(circle_package.pi);
6.
7.     l_area := circle_package.get_area(5);
8.     DBMS_OUTPUT.PUT_LINE(l_area);
9.
10.    l_circumference := circle_package.get_circumference(5);
11.    DBMS_OUTPUT.PUT_LINE(l_circumference);
12.  END;
```

Evaluation Copy

Code Explanation

The code should be self-explanatory. Running this code will output:

```
3.1415926535897932
78.54
31.42
```

Note that if you try to call `circle_package.get_diameter()`, you will get an error:

```
component 'GET_DIAMETER' must be declared
```

Again, this function is private, and not available outside of the package.

Exercise 13: Modifying the Package

15 to 25 minutes

In this exercise, you will modify the `circle_package` package.

1. Open `circle-package.sql` and `circle-package-body.sql` in SQL Developer from the `Packages/Exercises` folder.
2. Make the `get_diameter()` function public.
3. Add three public functions to get the radius:
   A. `get_radius_from_diameter()`
      - *radius = diameter / 2*
   B. `get_radius_from_area()`
      - *radius = the square root of (area / pi)* - use the built-in `SQRT()` function.
   C. `get_radius_from_circumference()`
      - *radius = (circumference / pi) / 2*
4. Run the code in `Packages/Exercises/use-circle-package.sql` to test your solution. It should output:

```
Test 1 passed
Test 2 passed
Test 3 passed
```

   If it doesn't. Review your code and make sure your formulas are correct.

Evaluation Copy

EVALUATION COPY: Not to be used in class.

Solution: Packages/Solutions/circle-package.sql

```
1.   CREATE OR REPLACE PACKAGE circle_package
2.   IS
3.
4.    pi CONSTANT FLOAT := 3.1415926535897932;
5.
6.    FUNCTION get_area(radius_in NUMBER) RETURN NUMBER;
7.    FUNCTION get_circumference(radius_in NUMBER) RETURN NUMBER;
8.    FUNCTION get_diameter(radius_in NUMBER) RETURN NUMBER;
9.    FUNCTION get_radius_from_diameter(diameter_in NUMBER) RETURN NUMBER;
10.   FUNCTION get_radius_from_area(area_in NUMBER) RETURN NUMBER;
11.   FUNCTION get_radius_from_circumference(circumference_in NUMBER)
12.      RETURN NUMBER;
13.
14.  END circle_package;
```


EVALUATION COPY: Not to be used in class.

Solution: Packages/Solutions/circle-package-body.sql

```
1.   CREATE OR REPLACE PACKAGE BODY circle_package
2.   IS
     -------Lines 3 through 28 Omitted-------
29.    FUNCTION get_radius_from_diameter(diameter_in NUMBER)
30.      RETURN NUMBER
31.    IS
32.      -- nothing to declare
33.    BEGIN
34.      RETURN diameter_in / 2;
35.    END get_radius_from_diameter;
36.
37.    FUNCTION get_radius_from_area(area_in NUMBER)
38.      RETURN NUMBER
39.    IS
40.      -- nothing to declare
41.    BEGIN
42.      RETURN SQRT(area_in / pi);
43.    END get_radius_from_area;
44.
45.    FUNCTION get_radius_from_circumference(circumference_in NUMBER)
46.      RETURN NUMBER
47.    IS
48.      -- nothing to declare
49.    BEGIN
50.      RETURN (circumference_in / pi) / 2;
51.    END get_radius_from_circumference;
52.
53.  END circle_package;
```

Evaluation Copy

6.4. Building an Employee Package

Let's now build a package for working with employees. We'll break the package up into the following sections:

1. **CURSORS** - The cursors that our package contains.
2. **INSERT** - A function for inserting a new employee.
3. **GETTERS (RECORDS)** - Functions that return employee records.
4. **GETTERS (FIELDS)** - Functions that return field values from the `employees` table.

5. **SETTERS** - Procedures for updating fields in the `employees` table.
6. **VALIDATION** - Procedures for validating fields in the `employees` table.
7. **AUDITING** - Procedures for auditing changes related to employees.
8. **DELETE** - Function for deleting an employee.

We'll start by creating the package specification with:

1. An `INSERT` function.
2. A single getter function that takes an employee's `employee_id` and returns an employee record.
3. A `DELETE` function.

Demo 6.4: Packages/Demos/employee-package-1.sql

```
1.   CREATE OR REPLACE PACKAGE employee_package
2.   IS
3.     -- INSERT
4.     FUNCTION new_employee(
5.           first_name_in       IN employees.first_name%TYPE,
6.           last_name_in        IN employees.last_name%TYPE,
7.           email_in            IN employees.email%TYPE,
8.           phone_number_in     IN employees.phone_number%TYPE,
9.           job_id_in           IN employees.job_id%TYPE,
10.          salary_in           IN employees.salary%TYPE,
11.          manager_id_in       IN employees.manager_id%TYPE,
12.          department_id_in    IN employees.department_id%TYPE,
13.          commission_pct_in   IN employees.commission_pct%TYPE := NULL,
14.          hire_date_in        IN employees.hire_date%TYPE := SYSDATE
15.    )
16.      RETURN employees%ROWTYPE;
17.
18.    -- GETTERS (RECORDS)
19.    FUNCTION get_employee(employee_id_in IN employees.employee_id%TYPE)
20.      RETURN employees%ROWTYPE;
21.
22.    -- DELETE
23.    FUNCTION delete_employee(
24.                    employee_id_in IN employees.employee_id%TYPE
25.    )
26.      RETURN BOOLEAN;
27.
28.  END employee_package;
```

Evaluation Copy

EVALUATION COPY: Not to be used in class.

Code Explanation

These functions are declared:

1. `new_employee()` - for inserting a new employee.
2. `get_employee()` - for getting an employee record by `employee_id`.
3. `delete_employee()` - for delering an employee record by `employee_id`.

Evaluation Copy

And here is the related package body:

Demo 6.5: Packages/Demos/employee-package-body-1.sql

```
1.    CREATE OR REPLACE PACKAGE BODY employee_package
2.    IS
3.      -- INSERT
4.      FUNCTION new_employee(
5.            first_name_in       IN employees.first_name%TYPE,
6.            last_name_in        IN employees.last_name%TYPE,
7.            email_in            IN employees.email%TYPE,
8.            phone_number_in     IN employees.phone_number%TYPE,
9.            job_id_in           IN employees.job_id%TYPE,
10.           salary_in           IN employees.salary%TYPE,
11.           manager_id_in       IN employees.manager_id%TYPE,
12.           department_id_in    IN employees.department_id%TYPE,
13.           commission_pct_in   IN employees.commission_pct%TYPE := NULL,
14.           hire_date_in        IN employees.hire_date%TYPE := SYSDATE
15.     )
16.     RETURN employees%ROWTYPE
17.   IS
18.     l_employee_id employees.employee_id%TYPE;
19.     l_employee    employees%ROWTYPE;
20.   BEGIN
21.     -- Get new employee_id
22.     SELECT MAX(employee_id) + 1
23.     INTO l_employee_id
24.     FROM employees;
25.
26.     INSERT INTO employees (employee_id, first_name, last_name,
27.                            email, phone_number, hire_date,
28.                            job_id, salary, commission_pct,
29.                            manager_id, department_id)
30.     VALUES                (l_employee_id, first_name_in, last_name_in,
31.                            email_in, phone_number_in, hire_date_in,
32.                            job_id_in, salary_in, commission_pct_in,
33.                            manager_id_in, department_id_in)
34.     RETURNING employee_id, first_name, last_name,
35.               email, phone_number, hire_date,
36.               job_id, salary, commission_pct,
37.               manager_id, department_id INTO l_employee;
38.
39.     RETURN l_employee;
40.
41.   END new_employee;
42.
43.   -- GETTERS (RECORDS)
44.   FUNCTION get_employee(employee_id_in IN employees.employee_id%TYPE)
```

Evaluation Copy

EVALUATION COPY: Not to be used in class.

```
45.       RETURN employees%ROWTYPE
46.     IS
47.       l_employee employees%ROWTYPE;
48.     BEGIN
49.       SELECT *
50.       INTO l_employee
51.       FROM employees
52.       WHERE employee_id = employee_id_in;
53.
54.       RETURN l_employee;
55.
56.     EXCEPTION
57.       WHEN no_data_found THEN
58.         RETURN NULL;
59.
60.     END get_employee;
61.
62.     -- DELETE
63.     FUNCTION delete_employee(
64.       employee_id_in IN employees.employee_id%TYPE
65.     )
66.       RETURN BOOLEAN
67.     IS
68.     BEGIN
69.       DELETE
70.       FROM employees
71.       WHERE employee_id = employee_id_in;
72.
73.       RETURN SQL%FOUND;
74.
75.     END delete_employee;
76.
77.   END employee_package;
```

Evaluation Copy

Code Explanation

Things to notice:

1. In the `new_employee()` function, we need to find the maximum `employee_id` and add `1` to it to get the new `employee_id`. That's because the `employee_id` primary key was not generated as an `IDENTITY`. See the Oracle documentation[5] for more information on identity columns.

5. https://docs.oracle.com/en/database/oracle/oracle-database/19/sqlrf/CREATE-TABLE.html

2. The `new_employee()` function returns the newly created employee record.
3. In the `get_employee()` function we return `NULL` on a `no_data_found` exception, leaving it up to the calling code to decide what to do if there is no employee with the passed-in `employee_id`.
4. The `delete_employee()` function just deletes the employee record from the `employees` table and returns `SQL%FOUND`, which means it returns `TRUE` if a record was deleted and `FALSE` if not. Note that this function does not take into account parent-child relationships. It is entirely possible that the delete fails because the `employee_id` exists in another table as a foreign key. In that case, you will get an error similar to the following:

Evaluation Copy

```
ORA-02292: integrity constraint (HR.JHIST_EMP_FK) violated - child record found
```

The calling code could decide how to handle this exception.

The following code shows how to use our new package:

EVALUATION COPY: Not to be used in class.

Demo 6.6: Packages/Demos/use-employee-package-1.sql

```
1.   DECLARE
2.     l_employee_id employees.employee_id%TYPE;
3.     l_employee_1  employees%ROWTYPE;
4.     l_employee_2  employees%ROWTYPE;
5.   BEGIN
6.     -- Insert employee
7.     l_employee_1 := employee_package.new_employee(
8.           first_name_in       => 'Nat',
9.           last_name_in        => 'Dunn',
10.          email_in            => 'NDUNN',
11.          phone_number_in     => '555-054-2211',
12.          job_id_in           => 'IT_PROG',
13.          salary_in           => 4001,
14.          manager_id_in       => 103,
15.          department_id_in    => 60
16.    );
17.
18.    DBMS_OUTPUT.PUT_LINE(l_employee_1.first_name || ' ' ||
19.                      l_employee_1.last_name || ' inserted with id ' ||
20.                      l_employee_1.employee_id || '.');
21.
22.    -- Get a different employee
23.    l_employee_2 := employee_package.get_employee(100);
24.    DBMS_OUTPUT.PUT_LINE('Employee is ' || l_employee_2.first_name ||
25.                         ' ' || l_employee_2.last_name || '.');
26.  END;
27.
28.  -- Delete the record (You may need to change the id)
29.  DECLARE
30.    l_employee_id employees.employee_id%TYPE := 207;
31.    l_employee    employees%ROWTYPE;
32.    l_result      BOOLEAN;
33.  BEGIN
34.    l_employee  := employee_package.get_employee(l_employee_id);
35.    l_result    := employee_package.delete_employee(l_employee_id);
36.    IF l_result THEN
37.      DBMS_OUTPUT.PUT_LINE(l_employee.first_name || ' ' ||
38.                           l_employee.last_name ||' deleted.');
39.    ELSE
40.      DBMS_OUTPUT.PUT_LINE('No employee with id ' ||
41.                           l_employee_id ||
42.                           ' to delete.');
43.    END IF;
44.  END;
```

Evaluation Copy

Code Explanation

1. Run the first block of code to insert a new employee and then, separately, get the employee with `employee_id 100` (this isn't the one you just inserted). It should output:

```
Nat Dunn inserted with id 207.
Employee is Steven King.
```

Evaluation Copy

2. Run the second block to delete the new record you just inserted. You may need to change the id to match the one returned when you ran the top block. It should output:

```
Nat Dunn deleted.
```


EVALUATION COPY: Not to be used in class.

Exercise 14: Adding a get_manager() Function

15 to 25 minutes

In this exercise, you will add a `get_manager()` function to `employee_package`.

1. Open `Packages/Exercises/employee-package-2.sql` in SQL Developer.
2. Declare two new functions:
    A. `get_manager()` that takes an `employee_id` parameter and returns an employee record. If you need help with the SQL statement use `Packages/Exercises/get-manager-select.sql` for guidance.
    B. `get_hire_date()` that takes an `employee_id` parameter and returns the date that the employee was hired.
3. Open `Packages/Exercises/employee-package-body-2.sql` in SQL Developer.
4. Define the two functions:
    A. `get_manager()` should return one of the following:
        i. The employee record of the manager of the employee with the passed-in `employee_id`.
        ii. `NULL` if the employee with the passed-in `employee_id` has no manager (i.e., there is no data found by the `SELECT` statement).

    If there is no employee record with the passed-in id, it should raise a `no_such_employee` exception. Note that you can use the package's `get_employee()` function to check that. For example:

```
l_employee := get_employee(employee_id_in);
IF l_employee.employee_id IS NULL THEN
  RAISE no_such_employee;
END IF;
```

    B. `get_hire_date()` should return the `hire_date` of the employee with the passed-in `employee_id` or `NULL` if there is no employee with the passed-in `employee_id`.
5. Run the code in both files to create the package.

Evaluation Copy

6. Test the package by running the code in `Packages/Exercises/use-employee-package-2.sql`. It should output:

```
Manager is Lex De Haan.
Employee hired on 01/03/2006.
```

Evaluation Copy

7. Change the value of `l_employee_id` to `100`. This is Steven King's `employee_id`. Steven King has no manager. Run the code and you should get:

```
Employee has no manager.
Employee hired on 06/17/2003.
```


EVALUATION COPY: Not to be used in class.

Solution: Packages/Solutions/employee-package-2.sql

```
1.    CREATE OR REPLACE PACKAGE employee_package
      -------Lines 2 through 21 Omitted-------
22.     FUNCTION get_manager(employee_id_in IN employees.employee_id%TYPE)
23.       RETURN employees%ROWTYPE;
24.
25.     -- GETTERS (FIELDS)
26.     FUNCTION get_hire_date(
27.                     employee_id_in IN employees.employee_id%TYPE
28.     )
29.       RETURN employees.hire_date%TYPE;
      -------Lines 30 through 37 Omitted-------
```

Evaluation Copy

EVALUATION COPY: Not to be used in class.

Solution: Packages/Solutions/employee-package-body-2.sql

```
1.   CREATE OR REPLACE PACKAGE BODY employee_package
     -------Lines 2 through 61 Omitted-------
62.    FUNCTION get_manager(employee_id_in IN employees.employee_id%TYPE)
63.      RETURN employees%ROWTYPE
64.    IS
65.      l_employee        employees%ROWTYPE;
66.      l_manager         employees%ROWTYPE;
67.      no_such_employee  EXCEPTION;
68.    BEGIN
69.
70.      l_employee := get_employee(employee_id_in);
71.      IF l_employee.employee_id IS NULL THEN
72.        RAISE no_such_employee;
73.      END IF;
74.
75.      SELECT *
76.      INTO l_manager
77.      FROM employees
78.      WHERE employee_id = (
79.        SELECT manager_id
80.        FROM employees
81.        WHERE employee_id = employee_id_in);
82.
83.      RETURN l_manager;
84.
85.    EXCEPTION
86.      WHEN no_such_employee THEN
87.        RAISE_APPLICATION_ERROR(-20102,
88.                                'No such employee.');
89.      WHEN no_data_found THEN
90.        RETURN NULL;
91.
92.    END get_manager;
93.
94.    -- GETTERS (FIELDS)
95.    FUNCTION get_hire_date(
96.                    employee_id_in IN employees.employee_id%TYPE
97.    )
98.      RETURN employees.hire_date%TYPE
99.    IS
100.     l_hire_date employees.hire_date%TYPE;
101.   BEGIN
102.     SELECT hire_date
103.     INTO l_hire_date
```

Evaluation Copy

```
104.      FROM employees
105.      WHERE employee_id = employee_id_in;
106.
107.      RETURN l_hire_date;
108.
109.    EXCEPTION
110.      WHEN no_data_found THEN
111.        RETURN NULL;
112.
113.    END get_hire_date;
      -------Lines 114 through 130 Omitted-------
```

Solution: Packages/Solutions/use-employee-package-2.sql

```
1.    DECLARE
2.      l_employee_id employees.employee_id%TYPE;
3.      l_manager     employees%ROWTYPE;
4.      l_hire_date   employees.hire_date%TYPE;
5.    BEGIN
6.      l_employee_id := 103;
7.
8.      -- Get employee's manager
9.      l_manager := employee_package.get_manager(l_employee_id);
10.     IF l_manager.employee_id IS NULL THEN
11.       DBMS_OUTPUT.PUT_LINE('Employee has no manager.');
12.     ELSE
13.       DBMS_OUTPUT.PUT_LINE('Manager is ' || l_manager.first_name ||
14.                            ' ' || l_manager.last_name || '.');
15.     END IF;
16.
17.     -- Get employee's hire date
18.     l_hire_date := employee_package.get_hire_date(l_employee_id);
19.     DBMS_OUTPUT.PUT_LINE('Employee hired on ' ||
20.                          TO_CHAR(l_hire_date, 'MM/DD/YYYY') || '.');
21.
22.   END;
```

Evaluation Copy

Code Explanation

Notice the `IF` condition:

```
IF l_manager.employee_id IS NULL THEN
```


EVALUATION COPY: Not to be used in class.

You may, understandably, be tempted to check if the whole record is `NULL`, like this:

```
IF l_manager IS NULL THEN
```

However, a record cannot be `NULL`. So, you must check a field within the record. Our practice is to check the primary key as we did in the code above.

6.5. Overloading Subprograms

It's possible to have two or more subprograms with the same name in a package; however, the functions must be differentiable by their formal parameters. For example, you **cannot** have the following two functions in the same package:

```
FUNCTION get_radius(diameter_in NUMBER) RETURN NUMBER;
FUNCTION get_radius(area_in NUMBER) RETURN NUMBER;
```

Evaluation Copy

PL/SQL would not be able to understand your intention when you call `get_radius(25)`. Is `25` the area or the diameter?

For PL/SQL to be able to differentiate between two subprograms of the same name, the formal parameters must differ either in number or in *family type* (e.g., numeric, character, date).

Overloading and Numeric Types

It is possible to have two different subprograms with the same name that are differentiated based on different subtypes of numeric parameters. For example, one subprogram could take a float and the other an integer. PL/SQL will match the most precise one it can. But be careful writing code that relies on this as it could easily cause confusion.

Consider the following three function declarations:

```
FUNCTION get_employee(employee_id_in employees.employee_id%TYPE)
  RETURN employees%ROWTYPE;

FUNCTION get_employee(email_in employees.email%TYPE)
  RETURN employees%ROWTYPE;

FUNCTION get_employee(first_name_in employees.first_name%TYPE,
                      last_name_in  employees.last_name%TYPE)
  RETURN employees%ROWTYPE;
```

1. The first function takes one parameter: a number type.
2. The second function takes one parameter: a character type.
3. The third function takes two parameters: both character types.

Evaluation Copy

All three of these functions return a record from the `employees` table as specified by `employees%ROWTYPE`.

Calls to these functions are shown below:

```
get_employee(101); -- employee_id
get_employee('NKOCHHAR'); -- email
get_employee('Neena', 'Kochhar'); -- first and last name
```


EVALUATION COPY: Not to be used in class.

Exercise 15: Adding Overloaded Functions to the Package

15 to 25 minutes

In this exercise, you will add overloaded `get_employee()` functions to `employee_package`.

1. Open `Packages/Exercises/employee-package-3.sql` in SQL Developer.
2. Declare two new `get_employee()` functions: one that accepts an email address and one that accepts first and last names. Both should return an employee record or `NULL` if no matching employee is found.
3. Open `Packages/Exercises/employee-package-body-3.sql` in SQL Developer.
4. Define the two functions that you just declared in the specification.
5. Run the code in both files to re-create the package.
6. Test the package by running the code in `Packages/Exercises/use-employee-package-3.sql`. It should output:

```
ID: 101, Name: Neena Kochhar., Email: NKOCHHAR.
ID: 101, Name: Neena Kochhar., Email: NKOCHHAR.
ID: 101, Name: Neena Kochhar., Email: NKOCHHAR.
```

Evaluation Copy

Solution: Packages/Solutions/employee-package-3.sql

```
1.    CREATE OR REPLACE PACKAGE employee_package
      -------Lines 2 through 17 Omitted-------
18.     -- GETTERS (RECORDS)
19.     FUNCTION get_employee(employee_id_in IN employees.employee_id%TYPE)
20.       RETURN employees%ROWTYPE;
21.
22.     FUNCTION get_employee(email_in IN employees.email%TYPE)
23.       RETURN employees%ROWTYPE;
24.
25.     FUNCTION get_employee(first_name_in   IN employees.first_name%TYPE,
26.                           last_name_in    IN employees.last_name%TYPE)
27.       RETURN employees%ROWTYPE;
      -------Lines 28 through 44 Omitted-------
```


EVALUATION COPY: Not to be used in class.

Solution: Packages/Solutions/employee-package-body-3.sql

```
1.    CREATE OR REPLACE PACKAGE BODY employee_package
      -------Lines 2 through 42 Omitted-------
43.     -- GETTERS (RECORDS)
44.     FUNCTION get_employee(employee_id_in IN employees.employee_id%TYPE)
45.       RETURN employees%ROWTYPE
46.     IS
47.       l_employee employees%ROWTYPE;
48.     BEGIN
49.       SELECT *
50.       INTO l_employee
51.       FROM employees
52.       WHERE employee_id = employee_id_in;
53.
54.       RETURN l_employee;
55.
56.     EXCEPTION
57.       WHEN no_data_found THEN
58.         RETURN NULL;
59.
60.     END get_employee;
61.
62.     FUNCTION get_employee(email_in IN employees.email%TYPE)
63.       RETURN employees%ROWTYPE
64.     IS
65.       l_employee employees%ROWTYPE;
66.     BEGIN
67.       SELECT *
68.       INTO l_employee
69.       FROM employees
70.       WHERE email = email_in;
71.
72.       RETURN l_employee;
73.
74.     EXCEPTION
75.       WHEN no_data_found THEN
76.         RETURN NULL;
77.
78.     END get_employee;
79.
80.     FUNCTION get_employee(first_name_in   IN employees.first_name%TYPE,
81.                           last_name_in    IN employees.last_name%TYPE)
82.       RETURN employees%ROWTYPE
83.     IS
84.       l_employee employees%ROWTYPE;
```

Evaluation Copy

```
85.     BEGIN
86.       SELECT *
87.       INTO l_employee
88.       FROM employees
89.       WHERE first_name = first_name_in
90.         AND last_name = last_name_in;
91.
92.       RETURN l_employee;
93.
94.     EXCEPTION
95.       WHEN no_data_found THEN
96.         RETURN NULL;
97.
98.     END get_employee;
      -------Lines 99 through 168 Omitted-------
```

6.6. Auditing

Evaluation Copy

The `job_history` table is used to audit employees' past jobs. Let's look at that table and the `employees` and `jobs` tables to get some information on Jonathan Taylor:

Demo 6.7: Packages/Demos/jonathan-taylor-job.sql

```
1.    SELECT  e.employee_id, e.hire_date, e.job_id AS current_job,
2.            jh.department_id, jh.start_date, jh.end_date,
3.            j.job_id, j.job_title
4.    FROM employees e
5.      JOIN job_history jh ON e.employee_id = jh.employee_id
6.      JOIN jobs j ON j.job_id = jh.job_id
7.    WHERE e.first_name = 'Jonathon' AND e.last_name = 'Taylor';
```

This will return:

	EMPLOYEE_ID	HIRE_DATE	CURRENT_JOB	DEPARTMENT_ID	START_DATE	END_DATE	JOB_ID	JOB_TITLE
1	176	24-MAR-06	SA_REP	80	24-MAR-06	31-DEC-06	SA_REP	Sales Representative
2	176	24-MAR-06	SA_REP	80	01-JAN-07	31-DEC-07	SA_MAN	Sales Manager

From this query, we can gather that Jonathon Taylor:

1. Was hired on March 24, 2006.
2. Worked as a sales representative (`SA_REP`) from March 24, 2006 to December 31, 2006.

EVALUATION COPY: Not to be used in class.

3. Worked as a sales manager (SA_MAN) from January 1, 2007 to December 31, 2007.
4. Is currently a sales representative.

In summary, Jonathan Taylor began as a sales representative, became a sales manager in less than a year, and then a year later once again became a sales representative.

If Jonathan Taylor were once again promoted to sales manager, we would update his record in the `employees` table and insert a new record to record his former job in the `job_history` table. Those queries are shown below:

```
-- Change job in employees
UPDATE employees
SET job_id = 'SA_MAN'
WHERE employee_id = 176;

-- Insert old job into job_history
INSERT INTO job_history
(employee_id, job_id, department_id, start_date, end_date)
VALUES (176, 'SA_REP', 80, '01-JAN-08', SYSDATE);
```

Evaluation Copy

You might notice that there is potential for error here. Someone could forget to insert the record into the `job_history` table. Or they might insert a record into the `job_history` table without updating the `employees` table. In an effort to prevent those errors, we'll add a `change_job()` procedure to `employee_package`. It will take care of both changing the job in the `employees` table and adding the record to the `job_history` table.

The procedure definition in the specification will look like this:

Demo 6.8: Packages/Demos/employee-package-4.sql

```
1.    CREATE OR REPLACE PACKAGE employee_package
      -------Lines 2 through 37 Omitted-------
38.     -- SETTERS (UPDATES)
39.     PROCEDURE change_job(
40.       employee_id_in    employees.employee_id%TYPE,
41.       job_id_in         employees.job_id%TYPE,
42.       department_id_in  employees.department_id%TYPE
43.     );
      -------Lines 44 through 51 Omitted-------
```

And here's the package body with both the `change_job()` procedure definition and a new private procedure for adding the record to `job_history`:

Evaluation Copy

EVALUATION COPY: Not to be used in class.

Demo 6.9: Packages/Demos/employee-package-body-4.sql

```
1.   CREATE OR REPLACE PACKAGE BODY employee_package
2.   IS
3.     /* PRIVATE */
4.     -- AUDITING
5.     PROCEDURE add_job_history(
6.       employee_id_in    job_history.employee_id%TYPE,
7.       start_date_in     job_history.start_date%TYPE,
8.       end_date_in       job_history.end_date%TYPE,
9.       job_id_in         job_history.job_id%TYPE,
10.      department_id_in  job_history.department_id%TYPE
11.    )
12.    IS
13.    BEGIN
14.
15.      INSERT INTO job_history
16.      (employee_id, start_date, end_date, job_id, department_id)
17.      VALUES( employee_id_in, start_date_in,
18.              end_date_in, job_id_in, department_id_in);
19.    END add_job_history;
20.
     -------Lines 21 through 169 Omitted-------
170.   -- SETTERS (UPDATES)
171.   PROCEDURE change_job(
172.     employee_id_in    employees.employee_id%TYPE,
173.     job_id_in         employees.job_id%TYPE,
174.     department_id_in  employees.department_id%TYPE
175.   )
176.   IS
177.     l_old_job_id          job_history.job_id%TYPE;
178.     l_old_department_id   job_history.department_id%TYPE;
179.     l_start_date          job_history.start_date%TYPE;
180.     l_end_date            job_history.end_date%TYPE := SYSDATE;
181.   BEGIN
182.
183.     -- Store old job_id and department_id in local variables
184.     SELECT job_id, department_id
185.     INTO l_old_job_id, l_old_department_id
186.     FROM employees
187.     WHERE employee_id = employee_id_in;
188.
189.     -- For start date of old job get day after end date of prior job
190.     SELECT MAX(end_date) + 1 -- Plus 1 day
191.     INTO l_start_date
192.     FROM job_history
```

Evaluation Copy

```
193.      WHERE employee_id = employee_id_in;
194.
195.      -- If this is first job change,
196.      -- get hire date for start date of old job
197.      IF l_start_date IS NULL THEN
198.        SELECT hire_date
199.        INTO l_start_date
200.        FROM employees
201.        WHERE employee_id = employee_id_in;
202.      END IF;
203.
204.      -- Call add_job_history() procedure
205.      add_job_history(employee_id_in    => employee_id_in,
206.                      start_date_in     => l_start_date,
207.                      end_date_in       => l_end_date,
208.                      job_id_in         => l_old_job_id,
209.                      department_id_in  => l_old_department_id);
210.
211.      -- Finally, update employees
212.      UPDATE employees
213.      SET job_id = job_id_in,
214.          department_id = department_id_in
215.      WHERE employee_id = employee_id_in;
216.
217.    END change_job;
      -------Lines 218 through 234 Omitted-------
```

Evaluation Copy

Code Explanation

Things to notice:

1. In `change_job()`, we first gather information on the employee's current job and store the results in local variables.
2. We then pass those values as parameters to the private `add_job_history()` procedure, which inserts the new record in the `job_history` table.
3. By making the `add_job_history()` procedure private, we ensure that it won't be called directly from outside the package. While this doesn't prevent people from adding records to

EVALUATION COPY: Not to be used in class.

the `job_history` table using SQL `INSERT` statements, it does provide a handy PL/SQL solution for handling both the update and the auditing in one step:

```
employee_package.change_job(
  employee_id_in = 176,
  job_id_in = 'SA_MAN',
  department_id_in = 80
);
```

4. To make this even safer, we could set permissions to prevent developers from directly inserting records into the `job_history` table via `INSERT` statements and from updating an employee's `job_id` via `UPDATE`s on the `employees` table.

Below we have code for using the new `change_job()` procedure:

Demo 6.10: Packages/Demos/use-employee-package-4.sql

```
1.   DECLARE
2.     CURSOR employee_jh_cursor IS
3.       SELECT j.job_title, d.department_name, jh.start_date, jh.end_date
4.       FROM job_history jh
5.         JOIN jobs j ON j.job_id = jh.job_id
6.         JOIN departments d ON d.department_id = jh.department_id
7.       WHERE employee_id = 176
8.       ORDER BY start_date;
9.     l_employee_id     employees.employee_id%TYPE  := 176;
10.    l_employee        employees%ROWTYPE;
11.  BEGIN
12.    -- Get employee by id
13.    l_employee := employee_package.get_employee(l_employee_id);
14.
15.    -- Change job
16.    employee_package.change_job(l_employee_id, 'SA_REP', 80);
17.
18.    FOR l_job IN employee_jh_cursor LOOP
19.
20.      DBMS_OUTPUT.PUT_LINE('Job: ' || l_job.job_title ||
21.                           ', Department: ' || l_job.department_name ||
22.                           ', Start Date: ' || l_job.start_date ||
23.                           ', End Date: '  || l_job.end_date || '.');
24.    END LOOP;
25.
26.  END;
```

Evaluation Copy

Code Explanation

This will output something like:

```
Job: Sales Representative, Department: Sales, Start Date: 24-MAR-06, End Date: 31-DEC-
06.
Job: Sales Manager, Department: Sales, Start Date: 01-JAN-07, End Date: 31-DEC-07.
Job: Sales Representative, Department: Sales, Start Date: 01-JAN-08, End Date: 24-OCT-
19.
```

Notice the use of the cursor. We will soon see how to move that cursor into the package.

Evaluation Copy

6.7. Validation Procedures

Earlier, we created an `update_employee_manager()` procedure that was used to change an employee's `manager_id`. A simplified version is shown below:

EVALUATION COPY: Not to be used in class.

Demo 6.11: Packages/Demos/update-employee-manager.sql

```
1.   CREATE OR REPLACE PROCEDURE update_employee_manager(
2.     employee_id_in      IN    employees.employee_id%TYPE,
3.     manager_id_in       IN    employees.manager_id%TYPE
4.   ) IS
5.     -- Give l_managers_manager_id default so it isn't NULL
6.     l_managers_manager_id   employees.manager_id%TYPE := -1;
7.
8.     -- Default l_temp_employee_id to manager_id_in. Used to check that
9.     -- employee_id_in doesn't report to l_temp_employee_id's boss
10.    l_temp_employee_id    employees.employee_id%TYPE := manager_id_in;
11.
12.    manager_is_self       EXCEPTION;
13.    reciprocal_managers   EXCEPTION;
14.  BEGIN
15.    IF employee_id_in = manager_id_in THEN
16.      RAISE manager_is_self;
17.    END IF;
18.      -- Make sure employee is not reporting to an underling
19.
20.    WHILE l_managers_manager_id IS NOT NULL LOOP
21.      SELECT manager_id
22.      INTO l_managers_manager_id
23.      FROM employees
24.      WHERE employee_id = l_temp_employee_id;
25.
26.      IF l_managers_manager_id = employee_id_in THEN
27.        RAISE reciprocal_managers;
28.      END IF;
29.      -- Assign manager's manager id to l_temp_employee_id
30.      l_temp_employee_id := l_managers_manager_id;
31.
32.    END LOOP;
33.
34.    -- Set new manager
35.    UPDATE employees
36.    SET manager_id = manager_id_in
37.    WHERE employee_id = employee_id_in;
38.
39.  EXCEPTION
40.    WHEN manager_is_self THEN
41.      -- re-raise the error to be caught by calling code
42.      RAISE_APPLICATION_ERROR(-20101,
43.                      'Employee cannot be own manager.');
44.    WHEN reciprocal_managers THEN
```

Evaluation Copy

```
45.       -- re-raise the error to be caught by calling code
46.       RAISE_APPLICATION_ERROR(-20102,
47.                           'Employees cannot be managed by an underling.');
48.
49.   END update_employee_manager;
```

Code Explanation

This procedure raises exceptions when:

1. The employee reports to themself (`manager_is_self`).
2. The employee reports to an underling (`reciprocal_managers`).

Evaluation Copy

Notice that this procedure can be broken into two separate procedures:

1. A procedure that updates an employee's manager.
2. A procedure that validates an employee's manager.

The updating procedure would check that the change was valid by calling the validating procedure before making the change.

In the following exercise, you will add these two procedures to `employee_package`.

EVALUATION COPY: Not to be used in class.

Exercise 16: Adding a Validation Procedure

25 to 40 minutes

In this exercise, you will add a procedure for validating an employee's manager to `employee_package`. You will also add a procedure for updating an employee's manager.

1. Open `Packages/Exercises/employee-package-5.sql` in SQL Developer.
2. Add definitions for `change_manager()` and `check_manager()` procedures. Both procedures should take the same two `IN` parameters:
    A. `employee_id_in`
    B. `manager_id_in`
3. Open `Packages/Exercises/employee-package-body-5.sql` in SQL Developer.
4. Start by writing the `check_manager()` procedure using the validation portion of the code found in `Packages/Demos/update-employee-manager.sql`. Note that this procedure does nothing if there are no exceptions. If it does find a problem, it should raise either the `manager_is_self` or `reciprocal_managers` exception.
5. Now write the `change_manager()` procedure. It should start by calling the `check_manager()` procedure and passing in values for `employee_id_in` and `manager_id_in` like this:

    ```
    check_manager(employee_id_in, manager_id_in);
    ```

    The `check_manager()` function will raise an exception if there's a problem with the `employee_id-manager_id` combination. Below that check, write a SQL statement to update the `manager_id` for this (and only this!) employee.
6. Open `Packages/Exercises/use-employee-package-5.sql` in SQL Developer. Notice that both `l_employee_id` and `l_new_manager_id` are set to `102`. This is not allowed. Run the code and you should get an error based on the `manager_is_self` exception.
7. Change `l_employee_id` and `l_new_manager_id` to `102` and `105`, respectively. Run the code and you should get an error based on the `reciprocal_managers` exception.

Evaluation Copy

8. Change `l_employee_id` and `l_new_manager_id` to `103` and `100`, respectively. Run the code. You should get this output:

```
Old Manager: Lex De Haan
New Manager: Steven King
```

Evaluation Copy

9. Execute a `ROLLBACK` to undo these changes:

```
ROLLBACK;
```


EVALUATION COPY: Not to be used in class.

Solution: Packages/Solutions/employee-package-5.sql

```
1.    CREATE OR REPLACE PACKAGE employee_package
      -------Lines 2 through 44 Omitted-------
45.     PROCEDURE change_manager(
46.       employee_id_in    employees.employee_id%TYPE,
47.       manager_id_in     employees.manager_id%TYPE
48.     );
49.
50.     -- VALIDATION
51.     PROCEDURE check_manager(
52.       employee_id_in  IN      employees.employee_id%TYPE,
53.       manager_id_in   IN      employees.manager_id%TYPE
54.     );
      -------Lines 55 through 62 Omitted-------
```

Evaluation Copy

EVALUATION COPY: Not to be used in class.

Solution: Packages/Solutions/employee-package-body-5.sql

```
1.   CREATE OR REPLACE PACKAGE BODY employee_package
     -------Lines 2 through 218 Omitted-------
219.   PROCEDURE change_manager(
220.     employee_id_in      IN    employees.employee_id%TYPE,
221.     manager_id_in       IN    employees.manager_id%TYPE
222.   ) IS
223.   BEGIN
224.     check_manager(employee_id_in, manager_id_in);
225.
226.     -- Set new manager
227.     UPDATE employees
228.     SET manager_id = manager_id_in
229.     WHERE employee_id = employee_id_in;
230.   END change_manager;
231.
232.   -- VALIDATION
233.   PROCEDURE check_manager(
234.     employee_id_in  IN      employees.employee_id%TYPE,
235.     manager_id_in   IN      employees.manager_id%TYPE
236.   )
237.   IS
238.     l_managers_manager_id employees.manager_id%TYPE := -1;
239.     l_temp_employee_id   employees.employee_id%TYPE := manager_id_in;
240.     manager_is_self       EXCEPTION;
241.     reciprocal_managers   EXCEPTION;
242.   BEGIN
243.     IF employee_id_in = manager_id_in THEN
244.       RAISE manager_is_self;
245.     END IF;
246.
247.     WHILE l_managers_manager_id IS NOT NULL LOOP
248.       SELECT manager_id
249.       INTO l_managers_manager_id
250.       FROM employees
251.       WHERE employee_id = l_temp_employee_id;
252.
253.       IF l_managers_manager_id = employee_id_in THEN
254.         RAISE reciprocal_managers;
255.       END IF;
256.       l_temp_employee_id := l_managers_manager_id;
257.
258.     END LOOP;
259.   -- Made it through loop without finding match
260.
```

Evaluation Copy

```
261.    EXCEPTION
262.      WHEN manager_is_self THEN
263.        -- re-raise the error to be caught by calling code
264.        RAISE_APPLICATION_ERROR(-20101,
265.                                'Employee cannot be own manager.');
266.      WHEN reciprocal_managers THEN
267.        -- re-raise the error to be caught by calling code
268.        RAISE_APPLICATION_ERROR(-20102,
269.                            'Employees cannot report to underlings.');
270.
271.    END check_manager;
      -------Lines 272 through 288 Omitted-------
```

6.8. Package Cursors

To add a public cursor to a package, you declare the cursor in the specification with a RETURN statement, like this:

```
CURSOR employee_cursor
  RETURN employees%ROWTYPE;
```

Evaluation Copy

Then you define the cursor in the package body, like this:

```
CURSOR employee_cursor
  RETURN employees%ROWTYPE
IS
  SELECT *
  FROM employees
  ORDER BY hire_date DESC;
```

Then you can reference this cursor as you would any other package object. For example, you could write:

```
FOR l_employee IN employee_package.employee_cursor LOOP

  DBMS_OUTPUT.PUT_LINE('Name: ' || l_employee.first_name ||
                       ' ' || l_employee.last_name || '.');
END LOOP;
```


EVALUATION COPY: Not to be used in class.

It is also possible to declare and define the cursor in the package specification, in which case, the cursor would not be defined in the package body.

❖ 6.8.1. Passing Parameters to Cursors

Remember that you can pass parameters to your cursors. For example, to create a cursor of employees by department, you could do it like this:

Evaluation Copy

```
CURSOR employee_cursor (
  department_id_in IN employees.department_id%TYPE
)
   employees%ROWTYPE
IS
  SELECT *
  FROM employees
  WHERE department_id = department_id_in
  ORDER BY hire_date DESC;
```

Exercise 17: Adding a Cursor to the Package

20 to 30 minutes

In this exercise, you will declare and define a cursor called `employees_jh_cursor` in the package specification and then write code to make use of that cursor.

1. Open `Packages/Exercises/employee-package-6.sql` in SQL Developer.
2. Add a definition and declaration for the `employees_jh_cursor` with one parameter: `employee_id_in`. You should base this on the cursor from `Packages/Exercises/use-employee-package-6.sql`
3. Run the code to create the package.
4. Open `Packages/Exercises/use-employee-package-6.sql` in SQL Developer if it's not open already.
5. Modify the code to make use of the new package cursor.
6. Test your code. It should output something like:

Evaluation Copy

```
Job: Sales Representative, Department: Sales, Start Date: 24-MAR-06, End Date:
31-DEC-06.
Job: Sales Manager, Department: Sales, Start Date: 01-JAN-07, End Date: 31-DEC-
07.
Job: Sales Representative, Department: Sales, Start Date: 01-JAN-08, End Date:
24-OCT-19.
```

check constraint (HR.JHIST_DATE_INTERVAL) violated

If you run this more than once, you may get an error similar to the following:

```
ORA-02290: check constraint (HR.JHIST_DATE_INTERVAL) violated
```

This is because the `start_date` and `end_date` in a `job_history` record can not be the same. You can resolve it by deleting records with end dates less than a day old:

```
ORA-02290: check constraint (HR.JHIST_DATE_INTERVAL) violated
```


EVALUATION COPY: Not to be used in class.

This is because the `start_date` and `end_date` in a `job_history` record can not be the same. You can resolve it by deleting records with end dates less than a day old:

```
DELETE FROM job_history
WHERE end_date > SYSDATE-1;
```

Evaluation Copy

Solution: Packages/Solutions/employee-package-6.sql

```
1.    CREATE OR REPLACE PACKAGE employee_package
2.    IS
3.      -- CURSORS
4.      CURSOR employee_jh_cursor (
5.        employee_id_in IN employees.employee_id%TYPE
6.      )
7.      IS
8.        SELECT j.job_title, d.department_name, jh.start_date, jh.end_date
9.        FROM job_history jh
10.         JOIN jobs j ON j.job_id = jh.job_id
11.         JOIN departments d ON d.department_id = jh.department_id
12.       WHERE employee_id = employee_id_in
13.       ORDER BY start_date;
14.
      -------Lines 15 through 74 Omitted-------
```

Solution: Packages/Solutions/use-employee-package-6.sql

Evaluation Copy

```
1.    DECLARE
2.      l_employee_id     employees.employee_id%TYPE := 176;
3.      l_employee        employees%ROWTYPE;
4.    BEGIN
5.      -- Get employee by id
6.      l_employee := employee_package.get_employee(l_employee_id);
7.
8.      -- Change job
9.      employee_package.change_job(l_employee_id, 'SA_REP', 80);
10.
11.     FOR l_job IN employee_package.employee_jh_cursor(l_employee_id)
12.       LOOP
13.
14.       DBMS_OUTPUT.PUT_LINE('Job: '         || l_job.job_title ||
15.                            ', Department: ' || l_job.department_name ||
16.                            ', Start Date: ' || l_job.start_date ||
17.                            ', End Date: '   || l_job.end_date || '.');
18.     END LOOP;
19.
20.   END;
```


EVALUATION COPY: Not to be used in class.

6.9. Benefits of Packages

Packages have many benefits, including:

1. **Organization** - The primary benefit of packaging is that it creates an application in which all related code is available, making it easy for developers to reuse this code.
2. **Black boxed code** - Much of the implementation of a package can be hidden. For example, a developer using a package doesn't necessarily need to know if one package object relies on another package object. Objects in the package will automatically be available to each other. If the objects were not packaged, the developer would have to make sure that both were available.
3. **Performance** - Packages are loaded into memory the first time a package subprogram is run, making other subprograms in the package run more quickly.
4. **Shared variables and cursors** - Global variables and cursors can be shared by different objects in the package.
5. **Granting Privileges** - Grouping objects in a package allows you to grant rolls to individual developers to all objects within the package, rather than doing so on an object-by-object basis.

Evaluation Copy

Conclusion

In this lesson, you have learned to create and work with PL/SQL packages.

EVALUATION COPY: Not to be used in class.

LESSON 7
Triggers

Topics Covered

☑ Triggers.

Introduction

In this lesson, you will learn to use triggers for validating data and auditing changes to the database.

7.1. What are Triggers?

Triggers are named blocks of code that run automatically when some event happens. Think: "When this happens it will *trigger* this." There are several types of triggers, but the most commonly used by developers are *Database Markup Language (DML) triggers*: triggers that fire when records are inserted, updated, or deleted. When we refer to triggers in this lesson, we are referring to DML triggers.

Evaluation Copy

❖ 7.1.1. Purpose of Triggers

While triggers can be used for different things, they are most commonly used to:

1. Peform validation on changes made to the database. If an exception occurs, the change that caused the trigger to fire will be rolled back.
2. Audit changes in the database.

7.2. Trigger Parts

A `CREATE OR REPLACE TRIGGER` statement is broken up as follows:

1. The create or replace line:

```
CREATE OR REPLACE TRIGGER trigger_name
```

2. An indication of whether the trigger should fire before or after the DML statement:

```
CREATE OR REPLACE TRIGGER trigger_name
BEFORE...
```

or

```
CREATE OR REPLACE TRIGGER trigger_name
AFTER...
```

3. The type(s) of DML statements and the table name that cause the trigger to fire. Some examples:

```
CREATE OR REPLACE TRIGGER trigger_name
BEFORE
  INSERT
ON employees...
```

```
CREATE OR REPLACE TRIGGER trigger_name
AFTER
  DELETE
ON employees...
```

Evaluation Copy

```
CREATE OR REPLACE TRIGGER trigger_name
BEFORE
  UPDATE
ON employees...
```

```
CREATE OR REPLACE TRIGGER trigger_name
AFTER
  INSERT
  OR UPDATE
ON employees...
```

```
CREATE OR REPLACE TRIGGER trigger_name
BEFORE
  INSERT
  OR UPDATE OF hire_date, salary
ON employees...
```


EVALUATION COPY: Not to be used in class.

Notice that with an `UPDATE`, you can specify which columns must be updated to cause the trigger to fire.

4. An indication of whether the trigger should fire once for the DML statement or once for every row affected by the DML statement. For example, if an `UPDATE` statement affects many rows, do you want the trigger to run just once before or after all the updates take place (a *statement-level* trigger), or do you want it to run when each row is updated (a *row-level* trigger)? The default is a statement-level trigger, but it is more common to create row-level triggers. To do so, add the `FOR EACH ROW` clause:

```
CREATE OR REPLACE TRIGGER trigger_name
BEFORE
  INSERT
  OR UPDATE OF hire_date, salary
ON employees
FOR EACH ROW...
```

Evaluation Copy

When using row-level triggers, the values of fields before the change can be accessed using `:OLD` and after the change using `:NEW`. For example:

   A. `:OLD.salary` would get the value of `salary` *before* the `UPDATE` or `DELETE`. For `INSERT`, `:OLD` values do not exist

   B. `:NEW.salary` would get the value of `salary` *after* the `UPDATE` or `INSERT`. For `DELETE`, `:NEW` values do not exist.

5. A `WHEN` clause to indicate any conditions that must be met for the trigger code to execute:

```
CREATE OR REPLACE TRIGGER trigger_name
BEFORE
  INSERT
  OR UPDATE OF hire_date, salary
ON employees
FOR EACH ROW
WHEN (conditions)...
```

6. The `DECLARE`, `BEGIN`, and `EXCEPTION` parts:

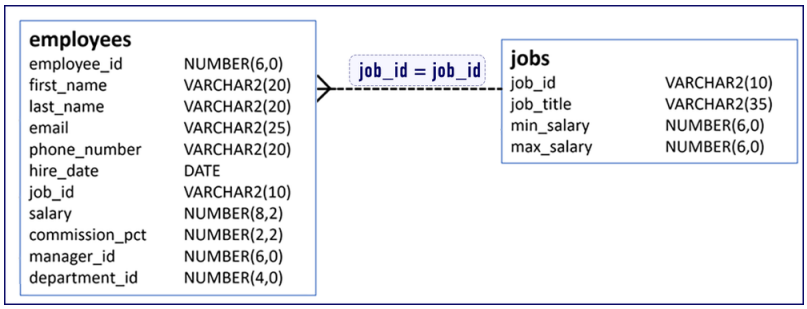
```
CREATE OR REPLACE TRIGGER trigger_name
BEFORE
  INSERT
  OR UPDATE OF hire_date, salary
ON employees
FOR EACH ROW
WHEN (conditions)
DECLARE
  -- declarations
BEGIN
  -- excutable statements
EXCEPTION
  -- exception handling
END trigger_name;
```

Let's take a look at some examples.

Evaluation Copy

7.3. Validation Triggers

Examine the two tables below:



Notice that the `jobs` table sets minimum and maximum salaries for each job, but there is no current rule to prevent someone from inserting new employees or updating existing ones with salaries that are outside the allowed salary range. It would be nice to do that with a check constraint[6] on the tables, but that's not possible as check constraints cannot refer to columns in other tables. But we can do the job with a trigger:

6. https://docs.oracle.com/en/database/oracle/oracle-database/19/sqlrf/constraint.html

EVALUATION COPY: Not to be used in class.

Demo 7.1: Triggers/Demos/employees-salary-biu-trigger.sql

```
1.   CREATE OR REPLACE TRIGGER employees_salary_biu_trigger
2.     BEFORE
3.       INSERT
4.       OR UPDATE OF salary, job_id
5.     ON employees
6.     FOR EACH ROW
7.   DECLARE
8.     l_min_salary jobs.min_salary%TYPE;
9.     l_max_salary jobs.max_salary%TYPE;
10.    invalid_salary EXCEPTION;
11.  BEGIN
12.
13.    SELECT min_salary, max_salary
14.    INTO l_min_salary, l_max_salary
15.    FROM jobs
16.    WHERE job_id = :NEW.job_id;
17.
18.    IF :NEW.salary NOT BETWEEN l_min_salary AND l_max_salary THEN
19.      RAISE invalid_salary;
20.    END IF;
21.
22.  EXCEPTION
23.    WHEN invalid_salary THEN
24.      -- re-raise the error to be caught by calling code
25.      RAISE_APPLICATION_ERROR(-20101,
26.                'Salary for ' || :NEW.job_id || ' must be between ' ||
27.                l_min_salary || ' and ' || l_max_salary || '.');
28.  END employees_salary_biu_trigger;
```

Evaluation Copy

Code Explanation

Things to notice:

1. The trigger fires in the following cases:
    A. A new record is inserted into the `employees` table.
    B. The `job_id` or `salary` of one or more existing records in the `employees` table is updated.

2. The values of the new or changed record are read using `:NEW`. For example, `:NEW.job_id`.

Run the code to create the trigger, then test the trigger by running the `UPDATE` statement in `Triggers/Demos/test-salary-trigger.sql`. It should output:

```
ORA-20101: Salary for AD_ASST must be between 3000 and 6000.
```

Now run the PL/SQL block in the same file. It should output:

```
ORA-20101: Salary for IT_PROG must be between 4000 and 10000.
```

Constraints vs. Triggers

Never use a trigger when the same job can be accomplished with a constraint. For example, you could use a contraint to set minimum and maximum values on an individual column that applies to all records in the table. In the trigger we just created, a constraint was not an option as there are different rules for different records depending on values found in a separate table.

Evaluation Copy

❖ 7.3.1. Naming Triggers

There are no hardset rules or conventions for naming triggers. Our convention is to join the following name parts on underscores:

1. Start with the table name possibly qualified by affected columns. For example: `employees_salary`.
2. Add `b` for `BEFORE` or `a` for `AFTER` followed by initials of the relevant DML statements: `i` for `INSERT`, `u` for `UPDATE`, `d` for `DELETE`. For example: `employees_salary_biu` for `BEFORE INSERT OR UPDATE`.
3. End with the word "trigger". For example: `employees_salary_biu_trigger` .

EVALUATION COPY: Not to be used in class.

Exercise 18: Creating a Trigger on the jobs Table

25 to 40 minutes

In this exercise, you will create a trigger on the `jobs` table that prevents changes that make an employee's salary out of the `min_salary` - `max_salary` range. This will only affect `UPDATE` statements.

1. Open `Triggers/Exercises/jobs-bu-trigger.sql` in SQL Developer. The file contains no code.
2. Create a trigger that fires when either the `min_salary` or `max_salary` is updated. The trigger should check all records in the `employees` table.
    - If the new `min_salary` is greater than any of the employee's who have that `job_id`, it should raise an `invalid_min_salary` exception and output something to the effect of:

```
ORA-20101: Error: Min salary of 4401 for AD_ASST is higher than salary
of Jennifer Whalen (200), which is 4400.
```

Evaluation Copy

    - If the new `max_salary` is less than any of the employee's who have that `job_id`, it should raise an `invalid_max_salary` exception and output something to the effect of:

```
ORA-20102: Error: Max salary of 4399 for AD_ASST is lower than salary of
 Jennifer Whalen (200), which is 4400.
```

Some Tips:

1. The trigger should fire *before* the record is updated.
2. One option is to use a cursor on the `employees` table and loop through it comparing the employee's salary to the new `min_salary` and then to the new `max_salary`. Remember that cursors can take parameters, so you might want to pass the `job_id` to this cursor.
3. You can store information about the employee (first and last name, id, and salary) in local variables as you iterate through the cursor. That way, if an exception occurs, those variables will hold information about the employee that caused the exception.

When you have finished, execute your code to create the trigger. If you get any errors, try to fix them and then execute your code again.

Once you have successfully created the trigger without errors, try running the following two queries. They should both cause exceptions as Jennifer Whalen, an administration assistant, has a salary of `4400`.

```
UPDATE jobs
SET min_salary = 4401
WHERE job_id = 'AD_ASST';
```

This should output:

Evaluation Copy

```
ORA-20101: Error: Min salary of 4401 for AD_ASST is higher than salary of Jennifer Whalen
 (200), which is 4400.
```

```
UPDATE jobs
SET max_salary = 4399
WHERE job_id = 'AD_ASST';
```

This should output:

```
ORA-20102: Error: Max salary of 4399 for AD_ASST is lower than salary of Jennifer Whalen
 (200), which is 4400.
```


EVALUATION COPY: Not to be used in class.

Solution: Triggers/Solutions/jobs-bu-trigger.sql

```
1.  CREATE OR REPLACE TRIGGER jobs_bu_trigger
2.    BEFORE
3.      UPDATE OF min_salary, max_salary
4.    ON jobs
5.    FOR EACH ROW
6.  DECLARE
7.    CURSOR employee_cursor (job_id_in jobs.job_id%TYPE) IS
8.      SELECT employee_id, first_name, last_name, salary
9.      FROM employees
10.     WHERE job_id = job_id_in;
11.   invalid_min_salary  EXCEPTION;
12.   invalid_max_salary  EXCEPTION;
13.   l_first_name        employees.first_name%TYPE;
14.   l_last_name         employees.last_name%TYPE;
15.   l_employee_id       employees.employee_id%TYPE;
16.   l_salary            employees.salary%TYPE;
17. BEGIN
18.
19.   FOR l_employee IN employee_cursor(:NEW.job_id) LOOP
20.     l_first_name  := l_employee.first_name;
21.     l_last_name   := l_employee.last_name;
22.     l_employee_id := l_employee.employee_id;
23.     l_salary      := l_employee.salary;
24.
25.     IF l_employee.salary < :NEW.min_salary THEN
26.       RAISE invalid_min_salary;
27.     ELSIF l_employee.salary > :NEW.max_salary THEN
28.       RAISE invalid_max_salary;
29.     END IF;
30.
31.   END LOOP;
32.
33. EXCEPTION
34.   WHEN invalid_min_salary THEN
35.     -- re-raise the error to be caught by calling code
36.     RAISE_APPLICATION_ERROR(-20101,
37.                   'Error: Min salary of ' || :NEW.min_salary ||
38.                   ' for ' || :NEW.job_id ||
39.                   ' is higher than salary of ' ||
40.                   l_first_name || ' ' || l_last_name ||
41.                   ' (' || l_employee_id || '), which is ' ||
42.                   l_salary || '.');
43.   WHEN invalid_max_salary THEN
44.     -- re-raise the error to be caught by calling code
```

Evaluation Copy

EVALUATION COPY: Not to be used in class.

```
45.       RAISE_APPLICATION_ERROR(-20102,
46.                       'Error: Max salary of ' || :NEW.max_salary ||
47.                       ' for ' || :NEW.job_id ||
48.                       ' is lower than salary of ' ||
49.                       l_first_name || ' ' || l_last_name ||
50.                       ' (' || l_employee_id || '), which is ' ||
51.                       l_salary || '.');
52.   END jobs_bu_trigger;
```

7.4. The WHEN Clause

The `WHEN` clause is used to indicate any conditions that must be met for the trigger code to execute. Note that the `WHEN` clause does not stop the trigger from firing, but the code following the `WHEN` clause will only run if the `WHEN` conditions are met. The syntax is as follows:

```
CREATE OR REPLACE TRIGGER trigger_name
BEFORE
  INSERT
  OR UPDATE OF column_names
ON table_name
FOR EACH ROW
WHEN (conditions)...
```

Some things to note:

1. The whole of the conditions must be contained within parentheses:

   ```
   WHEN (condition_1 OR condition_2)
   ```

   This is true even if there is only one condition:

   ```
   WHEN (condition)
   ```

2. You can reference `NEW` and `OLD` values, but in the `WHEN` clause you do not include the colon:

   ```
   WHEN (NEW.column_name != OLD.column_name)
   ```

Exercise 19: Using the WHEN Clause

5 to 10 minutes

In this exercise you will add a `WHEN` clause to `jobs_bu_trigger`. Note that the body of this trigger only needs to run if we are further restricting the allowed salary range (i.e., lowering the maximum salary or raising the minimum salary).

1. Open `Triggers/Exercises/jobs-bu-trigger.sql` in SQL Developer. This should contain your solution to the last exercise. If you weren't able to successfully complete that solution, open `jobs-bu-trigger.sql` from the `Solutions` folder and save it as `jobs-bu-trigger-2.sql` in the `Exercises` folder.
2. Add a `WHEN` clause, so that the trigger body only executes if the new `min_salary` is higher than the old `min_salary` or the new `max_salary` is lower than the old `max_salary`.
3. Add the following line to the beginning of the trigger body so that you know when the body of the trigger has executed:

```
DBMS_OUTPUT.PUT_LINE('Trigger code executed.');
```

Evaluation Copy

When you have finished, try running the following two queries:

```
UPDATE jobs
SET min_salary = 4401
WHERE job_id = 'AD_ASST';

UPDATE jobs
SET max_salary = 4399
WHERE job_id = 'AD_ASST';
```

As before, they should both cause exceptions as Jennifer Whalen, an administration assistant, has a salary of `4400`. In addition to the exception output, you should see the following line:

```
Trigger code executed.
```

This indicates that the body of the trigger was executed.

Now try running these updates:

EVALUATION COPY: Not to be used in class.

```
UPDATE jobs
SET min_salary = 2999
WHERE job_id = 'AD_ASST';

UPDATE jobs
SET max_salary = 6001
WHERE job_id = 'AD_ASST';
```

Evaluation Copy

The updates should work because they are not further restricting the salary range. If an employee's salary was lower than `6000` (the previous value of `max_salary`), then it will also be lower than `6001`. As such, there is no need to run the body of the trigger. If your `WHEN` clause is correct, the only output you will get for each of these is:

```
1 row updated.
```

Rollback the update.

Solution: Triggers/Solutions/jobs-bu-trigger-2.sql

```
1.   CREATE OR REPLACE TRIGGER jobs_bu_trigger
2.     BEFORE
3.       UPDATE OF min_salary, max_salary
4.     ON jobs
5.     FOR EACH ROW
6.   WHEN (NEW.min_salary > OLD.min_salary
7.         OR NEW.max_salary < OLD.max_salary)
8.   DECLARE
9.     CURSOR employee_cursor (job_id_in jobs.job_id%TYPE) IS
10.      SELECT employee_id, first_name, last_name, salary
11.      FROM employees
12.      WHERE job_id = job_id_in;
13.    invalid_min_salary  EXCEPTION;
14.    invalid_max_salary  EXCEPTION;
15.    l_first_name        employees.first_name%TYPE;
16.    l_last_name         employees.last_name%TYPE;
17.    l_employee_id       employees.employee_id%TYPE;
18.    l_salary            employees.salary%TYPE;
19.  BEGIN
20.    DBMS_OUTPUT.PUT_LINE('Trigger code executed.');
     -------Lines 21 through 54 Omitted-------
```

Evaluation Copy

7.5. Audit Triggers

Triggers are often used for auditing changes in the database. To illustrate, we'll create an `employees_audit` table:

EVALUATION COPY: Not to be used in class.

Demo 7.2: Triggers/Demos/create-employees-audit.sql

```
1.   DROP TABLE employees_audit; -- in case it exists already
2.
3.   CREATE TABLE employees_audit
4.   (
5.     employee_id     NUMBER(6) REFERENCES employees(employee_id),
6.     first_name      VARCHAR2(20),
7.     last_name       VARCHAR2(25),
8.     email           VARCHAR2(25),
9.     phone_number    VARCHAR2(20),
10.    hire_date       DATE,
11.    job_id          VARCHAR2(10),
12.    salary          NUMBER(8,2),
13.    commission_pct  NUMBER(2,2),
14.    manager_id      NUMBER(6) REFERENCES employees(employee_id),
15.    department_id   NUMBER(4) REFERENCES departments(department_id),
16.    operation       CHAR,
17.    change_date     DATE DEFAULT SYSDATE,
18.    change_made_by  VARCHAR2(20)
19.  );
```

Evaluation Copy

Code Explanation

This table has columns corresponding to the columns in the `employees` table, plus three new columns:

1. `operation` - the operation responsible for the change:
    - `I` for `INSERT`
    - `U` for `UPDATE`
    - `D` for `DELETE`
2. `change_date` - the date the change was made, defaulting to `SYSDATE`.
3. `change_made_by` - the user who made the change.

Next we will create the trigger, but before we do, one more thing to learn:

❖ 7.5.1. INSERTING, UPDATING, DELETING

Row-level triggers provide information on the operation that caused the trigger to fire via `INSERTING`, `UPDATING`, and `DELETING`, which are called *conditional predicates*. These hold values of `TRUE` or `FALSE`,

indicating whether or not that operation occurred. For auditing, we will generally store the `:NEW` values when `INSERTING` or `UPDATING` is `TRUE` and the `:OLD` values when `DELETING` is `TRUE`.

It is also possible to check whether a specific column is being updated via `UPDATING('column_name')`. For example: `UPDATING('job_id')`.

Evaluation Copy

Now for the trigger:

EVALUATION COPY: Not to be used in class.

Demo 7.3: Triggers/Demos/employees-aiud-trigger.sql

```
1.   CREATE OR REPLACE TRIGGER employees_aiud_trigger
2.     AFTER
3.       INSERT OR UPDATE OR DELETE
4.     ON employees
5.     FOR EACH ROW
6.   DECLARE
7.     l_operation CHAR := CASE
8.         WHEN UPDATING THEN 'U'
9.         WHEN DELETING THEN 'D'
10.        ELSE 'I' END;
11.  BEGIN
12.    IF INSERTING OR UPDATING THEN
13.      INSERT INTO employees_audit (
14.        employee_id,
15.        first_name,
16.        last_name,
17.        email,
18.        phone_number,
19.        hire_date,
20.        job_id,
21.        salary,
22.        commission_pct,
23.        manager_id,
24.        department_id,
25.        operation,
26.        change_date,
27.        change_made_by
28.      )
29.      VALUES (
30.        :NEW.employee_id,
31.        :NEW.first_name,
32.        :NEW.last_name,
33.        :NEW.email,
34.        :NEW.phone_number,
35.        :NEW.hire_date,
36.        :NEW.job_id,
37.        :NEW.salary,
38.        :NEW.commission_pct,
39.        :NEW.manager_id,
40.        :NEW.department_id,
41.        l_operation,
42.        SYSDATE,
43.        USER
44.      );
```

Evaluation Copy

```
45.     ELSE -- DELETING
46.       INSERT INTO employees_audit (
47.         employee_id,
48.         first_name,
49.         last_name,
50.         email,
51.         phone_number,
52.         hire_date,
53.         job_id,
54.         salary,
55.         commission_pct,
56.         manager_id,
57.         department_id,
58.         operation,
59.         change_date,
60.         change_made_by
61.       )
62.       VALUES (
63.         :OLD.employee_id,
64.         :OLD.first_name,
65.         :OLD.last_name,
66.         :OLD.email,
67.         :OLD.phone_number,
68.         :OLD.hire_date,
69.         :OLD.job_id,
70.         :OLD.salary,
71.         :OLD.commission_pct,
72.         :OLD.manager_id,
73.         :OLD.department_id,
74.         l_operation,
75.         SYSDATE,
76.         USER
77.       );
78.     END IF;
79.   END employees_aiud_trigger;
```

Evaluation Copy

Code Explanation

Things to notice:

1. We use a `CASE` expression to set the value of `l_operation`.
2. If the trigger is caused by `INSERTING` or `UPDATING`, we insert the `:NEW` values of the record into the audit table.

EVALUATION COPY: Not to be used in class.

3. If the trigger is caused by `DELETING`, we insert the `:OLD` values of the record into the audit table.
4. We insert `l_operation` into the `operation` field.
5. We insert the value of `SYSDATE` into the `change_date` field.
6. We insert the value of `USER` (the name of the user who performed the operation that fired the trigger) into the `change_made_by` field.

Test the trigger by running the `UPDATE` statement and then the PL/SQL block in `Triggers/Demos/test-employees-aiud-trigger.sql`. Then select all the records from the `employees_audit` table:

```
SELECT employee_id, first_name, last_name, job_id, salary,
            operation, change_date, change_made_by
FROM employees_audit;
```

Evaluation Copy

It should output:

	EMPLOYEE_ID	FIRST_NAME	LAST_NAME	JOB_ID	SALARY	OPERATION	CHANGE_DATE	CHANGE_MADE_BY
1	200	Jennifer	Whalen	AD_ASST	4500	U	25-OCT-19	C##HR
2	207	Tia	Rayl	IT_PROG	5100	I	25-OCT-19	C##HR

Rollback your changes.

7.6. Statement-level Triggers

Statement-level triggers are much less common than row-level triggers, but they can be useful when you need to perform validation across multiple rows in a table; for example, when you want to make sure that there is at least one employee who doesn't report to anyone - at least one head honcho:

Demo 7.4: Triggers/Demos/employees-aud-trigger.sql

```
1.   CREATE OR REPLACE TRIGGER employees_aud_trigger
2.     AFTER
3.       UPDATE OF manager_id
4.       OR DELETE
5.     ON employees
6.   DECLARE
7.     num_head_honchos        NUMBER(6,0);
8.     cannot_remove_boss      EXCEPTION;
9.   BEGIN
10.    SELECT COUNT(*)
11.    INTO num_head_honchos
12.    FROM employees
13.    WHERE manager_id IS NULL;
14.
15.    IF num_head_honchos = 0 THEN
16.      RAISE cannot_remove_boss;
17.    END IF;
18.
19.  EXCEPTION
20.    WHEN cannot_remove_boss THEN
21.      RAISE_APPLICATION_ERROR(-20101,
22.                              'Must have at least one head honcho.');
23.  END employees_aud_trigger;
```

Evaluation Copy

Code Explanation

Notice that this trigger does not include `FOR EACH ROW`. The body of the code runs only once, even if multiple rows are affected.

Run the code to create the trigger and then try running:

```
UPDATE employees
SET manager_id = 102
WHERE employee_id = 100;
```

You should get the following error:

```
ORA-20101: Must have at least one head honcho.
```

EVALUATION COPY: Not to be used in class.

7.7. Compound Triggers

A compound trigger is like a package, in which you can store multiple triggers that affect the same table. In this lesson, we have created three triggers on the `employees` table:

Evaluation Copy

1. `employees_salary_biu_trigger` - to validate an employee's salary.
2. `employees_aiud_trigger` - to audit changes in the `employees` table.
3. `employees_aud_trigger` - to check that there is at least one head honcho.

The following code combines these three triggers into a single compound trigger:

Demo 7.5: Triggers/Demos/employees-compound-trigger.sql

```
1.   CREATE OR REPLACE TRIGGER employees_compound_trigger
2.   FOR INSERT OR UPDATE OR DELETE ON employees
3.   COMPOUND TRIGGER
4.
5.     l_min_salary jobs.min_salary%TYPE;
6.     l_max_salary jobs.max_salary%TYPE;
7.     l_managers_manager_id   employees.manager_id%TYPE;
8.     l_operation CHAR := CASE
9.         WHEN UPDATING THEN 'U'
10.        WHEN DELETING THEN 'D'
11.        ELSE 'I' END;
12.    invalid_salary EXCEPTION;
13.    num_head_honchos        NUMBER(6,0);
14.    cannot_remove_boss      EXCEPTION;
15.
16.  BEFORE EACH ROW IS
17.  BEGIN
18.
19.    SELECT min_salary, max_salary
20.    INTO l_min_salary, l_max_salary
21.    FROM jobs
22.    WHERE job_id = :NEW.job_id;
23.
24.    IF :NEW.salary NOT BETWEEN l_min_salary AND l_max_salary THEN
25.      RAISE invalid_salary;
26.    END IF;
27.
28.  EXCEPTION
29.    WHEN invalid_salary THEN
30.      -- re-raise the error to be caught by calling code
31.      RAISE_APPLICATION_ERROR(-20101,
32.                'Salary for ' || :NEW.job_id || ' must be between ' ||
33.                l_min_salary || ' and ' || l_max_salary || '.');
34.  END BEFORE EACH ROW;
35.
36.  AFTER EACH ROW IS
37.  BEGIN
38.    IF INSERTING OR UPDATING THEN
     -------Lines 39 through 103 Omitted-------
104.   END IF;
105. END AFTER EACH ROW;
106.
107. AFTER STATEMENT IS
108. BEGIN
```

Evaluation Copy

EVALUATION COPY: Not to be used in class.

```
109.    SELECT COUNT(*)
110.    INTO num_head_honchos
111.    FROM employees
112.    WHERE manager_id IS NULL;
113.
114.    IF num_head_honchos = 0 THEN
115.      RAISE cannot_remove_boss;
116.    END IF;
117.
118.  EXCEPTION
119.    WHEN cannot_remove_boss THEN
120.      RAISE_APPLICATION_ERROR(-20101,
121.                              'Must have at least one head honcho.');
122.  END AFTER STATEMENT;
123.
124.  END employees_compound_trigger;
```

Code Explanation

Things to notice:

Evaluation Copy

1. There is a new `FOR` statement for indicating which types of statements fire the compound trigger:

   ```
   FOR INSERT OR UPDATE OR DELETE ON employees
   ```

2. The `FOR` statement is followed by `COMPOUND TRIGGER`.
3. All variables from the triggers are moved to the top of the compound trigger and will be global, meaning they can be used in any of the individual triggers.
4. The compound trigger is broken into four optional sections:
   A. `BEFORE STATEMENT IS` - runs once before *all rows* are changed. We don't use this in our compound trigger.
   B. `BEFORE EACH ROW IS` - runs once before *each row* is changed.
   C. `AFTER EACH ROW IS` - runs once after *each row* is changed.
   D. `AFTER STATEMENT IS` - runs once after *all rows* are changed.

Run the code to create the compound trigger.

After creating the compound trigger, drop the three triggers we created before:

```
DROP TRIGGER employees_salary_biu_trigger;
DROP TRIGGER employees_aiud_trigger;
DROP TRIGGER employees_aud_trigger;
```

You can run any of the same tests we've run earlier in the lesson to see that this works the same as when we had separate individual triggers. For example, run:

```
UPDATE employees
SET salary = 6001
WHERE employee_id = 200;
```

This will output:

```
ORA-20101: Salary for AD_ASST must be between 3000 and 6000.
ORA-06512: at "HR.EMPLOYEES_COMPOUND_TRIGGER", line 29
ORA-04088: error during execution of trigger 'HR.EMPLOYEES_COMPOUND_TRIGGER'
```

Evaluation Copy

7.8. Trigger Warning

While triggers can be useful, you should be careful not to overuse them or use them incorrectly. You should be particularly wary of using a trigger to change data (e.g., when this is updated, update this other record). The problem is that triggers run behind the scenes, so a developer writing new code that makes a change to the database can be unaware that the change will result in other changes caused by triggers. **When you create a trigger, be sure to think of how it might affect any future code.**

Conclusion

In this lesson, you have learned to use triggers to validate data and audit changes to the database.

EVALUATION COPY: Not to be used in class.