Oracle Database 10*g*: Develop PL/SQL Program Units

Volume 2 • Student Guide

D17169GC21

Edition 2.1

December 2006

D48231

ORACLE

Authors

Tulika Srivastava
Glenn Stokol

Technical Contributors and Reviewers

Chaitanya Koratamaddi
Dr. Christoph Burandt
Zarko Cesljas
Yanti Chang
Kathryn Cunningham
Burt Demchick
Laurent Dereac
Peter Driver
Bryan Roberts
Bryn Llewellyn
Nancy Greenberg
Craig Hollister
Thomas Hoogerwerf
Taj-Ul Islam
Inger Joergensen
Eric Lee
Malika Marghadi
Hildegard Mayr
Nagavalli Pataballa
Sunitha Patel
Srinivas Putrevu
Denis Raphaely
Helen Robertson
Grant Spencer
Glenn Stokol
Tone Thomas
Priya Vennapusa
Lex Van Der Werff

Graphic Designer

Satish Bettegowda

Editors

Nita Pavitran
Richard Wallis

Publisher

Sheryl Domingue

Contents

Preface

Profile

Before You Begin This Course

Before you begin this course, you should have thorough knowledge of SQL and *i*SQL*Plus, as well as working experience in developing applications. Prerequisites are any of the following Oracle University courses or combinations of courses:

- *Oracle Database 10g: Introduction to SQL*
- *Oracle Database 10g: SQL Fundamentals I* and *Oracle Database 10g: SQL Fundamentals II*
- *Oracle Database 10g: SQL and PL/SQL Fundamentals*
- *Oracle Database 10g: PL/SQL Fundamentals*

How This Course Is Organized

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

Related Publications

Oracle Publications

Title	**Part Number**
Oracle Database Application Developer's Guide – Fundamentals (10g Release 1)	*B10795-01*
Oracle Database Application Developer's Guide – Large Objects (10g Release 1)	*B10796-01*
PL/SQL Packages and Types Reference (10g Release 1)	*B10802-01*
PL/SQL User's Guide and Reference (10g Release 1)	*B10807-01*

Additional Publications

- System release bulletins
- Installation and user's guides
- *Read-me* files
- International Oracle Users Group (IOUG) articles
- *Oracle Magazine*

Typographic Conventions

Typographic Conventions in Text

Convention	Element	Example
Bold	Emphasized words and phrases in Web content only	To navigate within this application, do **not** click the Back and Forward buttons.
Bold italic	Glossary terms (if there is a glossary)	The ***algorithm*** inserts the new key.
Brackets	Key names	Press [Enter].
Caps and lowercase	Buttons, check boxes, triggers, windows	Click the Executable button. Select the Registration Required check box. Assign a When-Validate-Item trigger. Open the Master Schedule window.
Carets	Menu paths	Select File > Save.
Commas	Key sequences	Press and release these keys one at a time: [Alt], [F], [D]

Typographic Conventions (continued)

Typographic Conventions in Text (continued)

Convention	Object or Term	Example
Courier New, case sensitive	Code output, SQL and PL/SQL code elements, Java code elements, directory names, filenames, passwords, pathnames, URLs, user input, usernames	Code output: `debug.seti('I',300);` SQL code elements: Use the `SELECT` command to view information stored in the `last_name` column of the `emp` table. Java code elements: Java programming involves the `String` and `StringBuffer` classes. Directory names: `bin` (DOS), `$FMHOME` (UNIX) File names: Locate the `init.ora` file. Passwords: Use `tiger` as your password. Path names: Open `c:\my_docs\projects`. URLs: Go to `http://www.oracle.com`. User input: Enter `300`. Usernames: Log on as `scott`.
Initial cap	Graphics labels (unless the term is a proper noun)	Customer address (*but* Oracle Payables)
Italic	Emphasized words and phrases in print publications, titles of books and courses, variables	Do *not* save changes to the database. For further information, see *Oracle7 Server SQL Language Reference Manual*. Enter *`user_id@us.oracle.com`*, where *user_id* is the name of the user.
Plus signs	Key combinations	Press and hold these keys simultaneously: [Control] + [Alt] + [Delete]
Quotation marks	Lesson and chapter titles in cross references, interface elements with long names that have only initial caps	This subject is covered in Unit II, Lesson 3, "Working with Objects." Select the "Include a reusable module component" and click Finish. Use the "`WHERE` clause of query" property.

Typographic Conventions (continued)

Typographic Conventions in Navigation Paths

This course uses simplified navigation paths to direct you through Oracle applications, as in the following example.

Invoice Batch Summary

(N) Invoice > Entry > Invoice Batches Summary (M) Query > Find (B) Approve

This simplified path translates to the following sequence of steps:

1. (N) From the Navigator window, select Invoice > Entry > Invoice Batches Summary.
2. (M) From the menu, select Query > Find.
3. (B) Click the Approve button.

Notation:

(N) = Navigator	(I) = icon
(M) = menu	(H) = hyperlink
(T) = tab	(B) = button

Creating Triggers

Objectives

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

- **Describe the different types of triggers**
- **Describe database triggers and their uses**
- **Create database triggers**
- **Describe database trigger-firing rules**
- **Remove database triggers**

Lesson Aim

In this lesson, you learn how to create and use database triggers.

Types of Triggers

A trigger:

- Is a PL/SQL block or a PL/SQL procedure associated with a table, view, schema, or database
- Executes implicitly whenever a particular event takes place
- Can be either of the following:
  - Application trigger: Fires whenever an event occurs with a particular application
  - Database trigger: Fires whenever a data event (such as DML) or system event (such as logon or shutdown) occurs on a schema or database

Types of Triggers

Application triggers execute implicitly whenever a particular data manipulation language (DML) event occurs within an application. An example of an application that uses triggers extensively is an application developed with Oracle Forms Developer.

Database triggers execute implicitly when any of the following events occur:

- DML operations on a table
- DML operations on a view, with an `INSTEAD OF` trigger
- DDL statements, such as `CREATE` and `ALTER`

This is the case no matter which user is connected or which application is used. Database triggers also execute implicitly when some user actions or database system actions occur (for example, when a user logs on or the DBA shuts down the database).

Note: Database triggers can be defined on tables and on views. If a DML operation is issued on a view, then the `INSTEAD OF` trigger defines what actions take place. If these actions include DML operations on tables, then any triggers on the base tables are fired.

Database triggers can be system triggers on a database or a schema. For databases, triggers fire for each event for all users; for a schema, they fire for each event for that specific user.

This course explains how to create database triggers. Creating database triggers based on system events is discussed in the lesson titled "Applications for Triggers."

Guidelines for Designing Triggers

- You can design triggers to:
  - Perform related actions
  - Centralize global operations
- You must not design triggers:
  - Where functionality is already built into the Oracle server
  - That duplicate other triggers
- You can create stored procedures and invoke them in a trigger, if the PL/SQL code is very lengthy.
- The excessive use of triggers can result in complex interdependencies, which may be difficult to maintain in large applications.

ORACLE

Guidelines for Designing Triggers

- Use triggers to guarantee that related actions are performed for a specific operation.
- Use database triggers for centralized, global operations that should be fired for the triggering statement, independent of the user or application issuing the statement.
- Do not define triggers to duplicate or replace the functionality already built into the Oracle database. For example, implement integrity rules using declarative constraints, not triggers. To remember the design order for a business rule:
  - Use built-in constraints in the Oracle server, such as primary key, and so on.
  - Develop a database trigger or an application, such as a servlet or Enterprise JavaBeans (EJB) on your middle tier.
  - Use a presentation interface, such as Oracle Forms, HTML, JavaServer Pages (JSP) and so on, for data presentation rules.
- Excessive use of triggers can result in complex interdependencies, which may be difficult to maintain. Use triggers when necessary, and be aware of recursive and cascading effects.
- Avoid lengthy trigger logic by creating stored procedures or packaged procedures that are invoked in the trigger body.
- Database triggers fire for every user each time the event occurs on the trigger that is created.

Creating DML Triggers

Create DML statement or row type triggers by using:

```
CREATE [OR REPLACE] TRIGGER trigger_name
 timing
 event1 [OR event2 OR event3]
ON object_name
[[REFERENCING OLD AS old | NEW AS new]
 FOR EACH ROW
 [WHEN (condition)]]
trigger_body
```

- **A statement trigger fires once for a DML statement.**
- **A row trigger fires once for each row affected.**

Note: Trigger names must be unique with respect to other triggers in the same schema.

Creating DML Triggers

The components of the trigger syntax are:

- *trigger_name* uniquely identifies the trigger.
- *timing* indicates when the trigger fires in relation to the triggering event. Values are BEFORE, AFTER, and INSTEAD OF.
- event identifies the DML operation causing the trigger to fire.
  Values are INSERT, UPDATE [OF column], and DELETE.
- *object_name* indicates the table or view associated with the trigger.
- For row triggers, you can specify:
  - A REFERENCING clause to choose correlation names for referencing the old and new values of the current row (default values are OLD and NEW)
  - FOR EACH ROW to designate that the trigger is a row trigger
  - A WHEN clause to apply a conditional predicate, in parentheses, which is evaluated for each row to determine whether or not to execute the trigger body
- The *trigger_body* is the action performed by the trigger, implemented as either of the following:
  - An anonymous block with a DECLARE or BEGIN, and an END
  - A CALL clause to invoke a stand-alone or packaged stored procedure, such as:
    `CALL my_procedure;`

Types of DML Triggers

The trigger type determines whether the body executes for each row or only once for the triggering statement.

- A statement trigger:
  - Executes once for the triggering event
  - Is the default type of trigger
  - Fires once even if no rows are affected at all
- A row trigger:
  - Executes once for each row affected by the triggering event
  - Is not executed if the triggering event does not affect any rows
  - Is indicated by specifying the `FOR EACH ROW` clause

Types of DML Triggers

You can specify that the trigger will be executed once for every row affected by the triggering statement (such as a multiple row `UPDATE`) or once for the triggering statement, no matter how many rows it affects.

Statement Trigger

A statement trigger is fired once on behalf of the triggering event, even if no rows are affected at all. Statement triggers are useful if the trigger action does not depend on the data from rows that are affected or on data provided by the triggering event itself (for example, a trigger that performs a complex security check on the current user).

Row Trigger

A row trigger fires each time the table is affected by the triggering event. If the triggering event affects no rows, a row trigger is not executed. Row triggers are useful if the trigger action depends on data of rows that are affected or on data provided by the triggering event itself.

Note: Row triggers use correlation names to access the old and new column values of the row being processed by the trigger.

Trigger Timing

When should the trigger fire?

- `BEFORE`: Execute the trigger body before the triggering DML event on a table.
- `AFTER`: Execute the trigger body after the triggering DML event on a table.
- `INSTEAD OF`: Execute the trigger body instead of the triggering statement. This is used for views that are not otherwise modifiable.

Note: If multiple triggers are defined for the same object, then the order of firing triggers is arbitrary.

Trigger Timing

The **BEFORE** trigger timing is frequently used in the following situations:

- To determine whether the triggering statement should be allowed to complete (This eliminates unnecessary processing and enables a rollback in cases where an exception is raised in the triggering action.)
- To derive column values before completing an INSERT or UPDATE statement
- To initialize global variables or flags, and to validate complex business rules

The **AFTER** triggers are frequently used in the following situations:

- To complete the triggering statement before executing the triggering action
- To perform different actions on the same triggering statement if a BEFORE trigger is already present

The **INSTEAD OF** triggers provide a transparent way of modifying views that cannot be modified directly through SQL DML statements because a view is not always modifiable. You can write appropriate DML statements inside the body of an INSTEAD OF trigger to perform actions directly on the underlying tables of views.

Note: If multiple triggers are defined for a table, then the order in which multiple triggers of the same type fire is arbitrary. To ensure that triggers of the same type are fired in a particular order, consolidate the triggers into one trigger that calls separate procedures in the desired order.

Trigger-Firing Sequence

Use the following firing sequence for a trigger on a table when a single row is manipulated:

DML statement

```
INSERT INTO departments
   (department_id,department_name, location_id)
VALUES (400, 'CONSULTING', 2400);
```

Triggering action

→ `BEFORE` **statement trigger**

DEPARTMENT_ID	DEPARTMENT_NAME	LOCATION_ID
10	Administration	1700
20	Marketing	1800
30	Purchasing	1700
...		
400	CONSULTING	2400

→ `BEFORE` **row trigger**

→ `AFTER` **row trigger**

→ `AFTER` **statement trigger**

ORACLE

Trigger-Firing Sequence

Create a statement trigger or a row trigger based on the requirement that the trigger must fire once for each row affected by the triggering statement, or just once for the triggering statement, regardless of the number of rows affected.

When the triggering DML statement affects a single row, both the statement trigger and the row trigger fire exactly once.

Example

The SQL statement in the slide does not differentiate statement triggers from row triggers because exactly one row is inserted into the table using the syntax for the `INSERT` statement shown in the slide.

Trigger-Firing Sequence

Use the following firing sequence for a trigger on a table when many rows are manipulated:

```
UPDATE employees
  SET salary = salary * 1.1
  WHERE department_id = 30;
```

BEFORE statement trigger

EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID
114	Raphaely	30
115	Khoo	30
116	Baida	30
117	Tobias	30
118	Himuro	30
119	Colmenares	30

BEFORE row trigger
AFTER row trigger
...
BEFORE row trigger
AFTER row trigger
...
AFTER statement trigger

Trigger-Firing Sequence (continued)

When the triggering DML statement affects many rows, the statement trigger fires exactly once, and the row trigger fires once for every row affected by the statement.

Example

The SQL statement in the slide causes a row-level trigger to fire a number of times equal to the number of rows that satisfy the `WHERE` clause (that is, the number of employees reporting to department 30).

Trigger Event Types and Body

A trigger event:

- **Determines which DML statement causes the trigger to execute**
- **Types are:**
  - **`INSERT`**
  - **`UPDATE [OF column]`**
  - **`DELETE`**

A trigger body:

- **Determines what action is performed**
- **Is a PL/SQL block or a `CALL` to a procedure**

Triggering Event Types

The triggering event or statement can be an `INSERT`, `UPDATE`, or `DELETE` statement on a table.

- When the triggering event is an `UPDATE` statement, you can include a column list to identify which columns must be changed to fire the trigger. You cannot specify a column list for an `INSERT` or for a `DELETE` statement because it always affects entire rows.

```
. . . UPDATE OF salary . . .
```

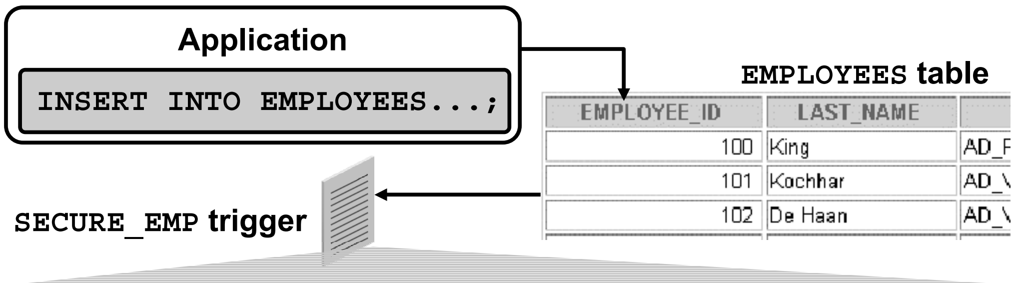
- The triggering event can contain one, two, or all three of these DML operations.

```
. . . INSERT or UPDATE or DELETE
. . . INSERT or UPDATE OF job_id . . .
```

The trigger body defines the action—that is, what needs to be done when the triggering event is issued. The PL/SQL block can contain SQL and PL/SQL statements, and can define PL/SQL constructs such as variables, cursors, exceptions, and so on. You can also call a PL/SQL procedure or a Java procedure.

Note: The size of a trigger cannot be greater than 32 KB.

Creating a DML Statement Trigger




```
CREATE OR REPLACE TRIGGER secure_emp
BEFORE INSERT ON employees BEGIN
 IF (TO_CHAR(SYSDATE,'DY') IN ('SAT','SUN')) OR
     (TO_CHAR(SYSDATE,'HH24:MI')
           NOT BETWEEN '08:00' AND '18:00') THEN
  RAISE_APPLICATION_ERROR(-20500, 'You may insert'
     ||' into EMPLOYEES table only during '
     ||' business hours.');
  END IF;
END;
```


Creating a DML Statement Trigger

In this example, the SECURE_EMP database trigger is a BEFORE statement trigger that prevents the INSERT operation from succeeding if the business condition is violated. In this case, the trigger restricts inserts into the EMPLOYEES table during certain business hours, Monday through Friday.

If a user attempts to insert a row into the EMPLOYEES table on Saturday, then the user sees an error message, the trigger fails, and the triggering statement is rolled back. Remember that the RAISE_APPLICATION_ERROR is a server-side built-in procedure that returns an error to the user and causes the PL/SQL block to fail.

When a database trigger fails, the triggering statement is automatically rolled back by the Oracle server

Testing SECURE_EMP

```
INSERT INTO employees (employee_id, last_name,
         first_name, email, hire_date,
         job_id, salary, department_id)
VALUES (300, 'Smith', 'Rob', 'RSMITH', SYSDATE,
        'IT_PROG', 4500, 60);
```

```
INSERT INTO employees (employee_id, last_name, first_name, email,
      *
ERROR at line 1:
ORA-20500: You may insert into EMPLOYEES table only during business hours.
ORA-06512: at "PLSQL.SECURE_EMP", line 4
ORA-04088: error during execution of trigger 'PLSQL.SECURE_EMP'
```


Testing SECURE_EMP

Insert a row into the EMPLOYEES table during nonbusiness hours. When the date and time are out of the business timings specified in the trigger, you receive the error message shown in the slide.

Using Conditional Predicates

```
CREATE OR REPLACE TRIGGER secure_emp BEFORE
INSERT OR UPDATE OR DELETE ON employees BEGIN
 IF (TO_CHAR(SYSDATE,'DY') IN ('SAT','SUN')) OR
   (TO_CHAR(SYSDATE,'HH24')
       NOT BETWEEN '08' AND '18') THEN
   IF DELETING THEN RAISE_APPLICATION_ERROR(
    -20502,'You may delete from EMPLOYEES table'||
           'only during business hours.');
   ELSIF INSERTING THEN RAISE_APPLICATION_ERROR(
    -20500,'You may insert into EMPLOYEES table'||
           'only during business hours.');
   ELSIF UPDATING('SALARY') THEN
    RAISE_APPLICATION_ERROR(-20503, 'You may '||
     'update SALARY only during business hours.');
   ELSE RAISE_APPLICATION_ERROR(-20504,'You may'||
     ' update EMPLOYEES table only during'||
     ' normal hours.');
   END IF;
 END IF;
END;
```


Combining Triggering Events

You can combine several triggering events into one by taking advantage of the special conditional predicates `INSERTING`, `UPDATING`, and `DELETING` within the trigger body.

Example

Create one trigger to restrict all data manipulation events on the `EMPLOYEES` table to certain business hours, Monday through Friday.

Creating a DML Row Trigger

```
CREATE OR REPLACE TRIGGER restrict_salary
BEFORE INSERT OR UPDATE OF salary ON employees
FOR EACH ROW
BEGIN
  IF NOT (:NEW.job_id IN ('AD_PRES', 'AD_VP'))
     AND :NEW.salary > 15000 THEN
    RAISE_APPLICATION_ERROR (-20202,
      'Employee cannot earn more than $15,000.');
  END IF;
END;
/
```


Creating a DML Row Trigger

You can create a BEFORE row trigger in order to prevent the triggering operation from succeeding if a certain condition is violated.

In the example, a trigger is created to allow certain employees to be able to earn a salary of more than 15,000. Suppose that a user attempts to execute the following UPDATE statement:

```
UPDATE employees
SET salary = 15500
WHERE last_name = 'Russell';
```

The trigger raises the following exception:

```
UPDATE EMPLOYEES
   *
ERROR at line 1:
ORA-20202: Employee cannot earn more than $15,000.
ORA-06512: at "PLSQL.RESTRICT_SALARY", line 5
ORA-04088: error during execution of trigger
"PLSQL.RESTRICT_SALARY"
```

Using `OLD` and `NEW` Qualifiers

```
CREATE OR REPLACE TRIGGER audit_emp_values
AFTER DELETE OR INSERT OR UPDATE ON employees
FOR EACH ROW
BEGIN
  INSERT INTO audit_emp(user_name, time_stamp, id,
    old_last_name, new_last_name, old_title,
    new_title, old_salary, new_salary)
  VALUES (USER, SYSDATE, :OLD.employee_id,
    :OLD.last_name, :NEW.last_name, :OLD.job_id,
    :NEW.job_id, :OLD.salary, :NEW.salary);
END;
/
```


Using `OLD` and `NEW` Qualifiers

Within a `ROW` trigger, reference the value of a column before and after the data change by prefixing it with the `OLD` and `NEW` qualifiers.

Data Operation	Old Value	New Value
`INSERT`	`NULL`	Inserted value
`UPDATE`	Value before update	Value after update
`DELETE`	Value before delete	`NULL`

Usage notes:

- The `OLD` and `NEW` qualifiers are available only in `ROW` triggers.
- Prefix these qualifiers with a colon (`:`) in every SQL and PL/SQL statement.
- There is no colon (`:`) prefix if the qualifiers are referenced in the `WHEN` restricting condition.

Note: Row triggers can decrease the performance if you perform many updates on larger tables.

Using OLD and NEW Qualifiers: Example Using AUDIT_EMP

```
INSERT INTO employees
 (employee_id, last_name, job_id, salary, ...)
VALUES (999, 'Temp emp', 'SA_REP', 6000,...);

UPDATE employees
 SET salary = 7000, last_name = 'Smith'
 WHERE employee_id = 999;
```

```
SELECT user_name, timestamp, ...
FROM audit_emp;
```

USER_NAME	TIME_STAMP	ID	OLD_LAST_NAME	NEW_LAST_NAME	OLD_TITLE	NEW_TITLE	OLD_SALARY	NEW_SALARY
ORA25	31-MAR-06			Temp emp		SA_REP		6000
ORA25	31-MAR-06	999	Temp emp	Smith	SA_REP	SA_REP	6000	7000

Using OLD and NEW Qualifiers: Example Using AUDIT_EMP

Create a trigger on the EMPLOYEES table to add rows to a user table, AUDIT_EMP, logging a user's activity against the EMPLOYEES table. The trigger records the values of several columns both before and after the data changes by using the OLD and NEW qualifiers with the respective column name.

There is an additional column named COMMENTS in AUDIT_EMP that is not shown in this slide.

Restricting a Row Trigger: Example

```
CREATE OR REPLACE TRIGGER derive_commission_pct
BEFORE INSERT OR UPDATE OF salary ON employees
FOR EACH ROW
WHEN (NEW.job_id = 'SA_REP')
BEGIN
 IF INSERTING THEN
   :NEW.commission_pct := 0;
 ELSIF :OLD.commission_pct IS NULL THEN
   :NEW.commission_pct := 0;
 ELSE
   :NEW.commission_pct := :OLD.commission_pct+0.05;
 END IF;
END;
/
```


Restricting a Row Trigger: Example

To restrict the trigger action to those rows that satisfy a certain condition, provide a `WHEN` clause.

Create a trigger on the `EMPLOYEES` table to calculate an employee's commission when a row is added to the `EMPLOYEES` table, or when an employee's salary is modified.

The `NEW` qualifier cannot be prefixed with a colon in the `WHEN` clause because the `WHEN` clause is outside the PL/SQL blocks.

Summary of the Trigger Execution Model

1. **Execute all `BEFORE STATEMENT` triggers.**
2. **Loop for each row affected:**
   a. **Execute all `BEFORE ROW` triggers.**
   b. **Execute the DML statement and perform integrity constraint checking.**
   c. **Execute all `AFTER ROW` triggers.**
3. **Execute all `AFTER STATEMENT` triggers.**

Note: Integrity checking can be deferred until the `COMMIT` operation is performed.

Trigger Execution Model

A single DML statement can potentially fire up to four types of triggers:

- `BEFORE` and `AFTER` statement triggers
- `BEFORE` and `AFTER` row triggers

A triggering event or a statement within the trigger can cause one or more integrity constraints to be checked. However, you can defer constraint checking until a `COMMIT` operation is performed.

Triggers can also cause other triggers—known as cascading triggers—to fire.

All actions and checks performed as a result of a SQL statement must succeed. If an exception is raised within a trigger and the exception is not explicitly handled, then all actions performed because of the original SQL statement are rolled back (including actions performed by firing triggers). This guarantees that integrity constraints can never be compromised by triggers.

When a trigger fires, the tables referenced in the trigger action may undergo changes by other users' transactions. In all cases, a read-consistent image is guaranteed for the modified values that the trigger needs to read (query) or write (update).

Implementing an Integrity Constraint with a Trigger

```
UPDATE employees SET department_id = 999
 WHERE employee_id = 170;
-- Integrity constraint violation error
```

```
CREATE OR REPLACE TRIGGER employee_dept_fk_trg
AFTER UPDATE OF department_id
ON employees FOR EACH ROW
BEGIN
 INSERT INTO departments VALUES(:new.department_id,
          'Dept '||:new.department_id, NULL, NULL);
EXCEPTION
   WHEN DUP_VAL_ON_INDEX THEN
    NULL; -- mask exception if department exists
END;
/
```

```
UPDATE employees SET department_id = 999
 WHERE employee_id = 170;
-- Successful after trigger is fired
```

ORACLE

Implementing an Integrity Constraint with a Trigger

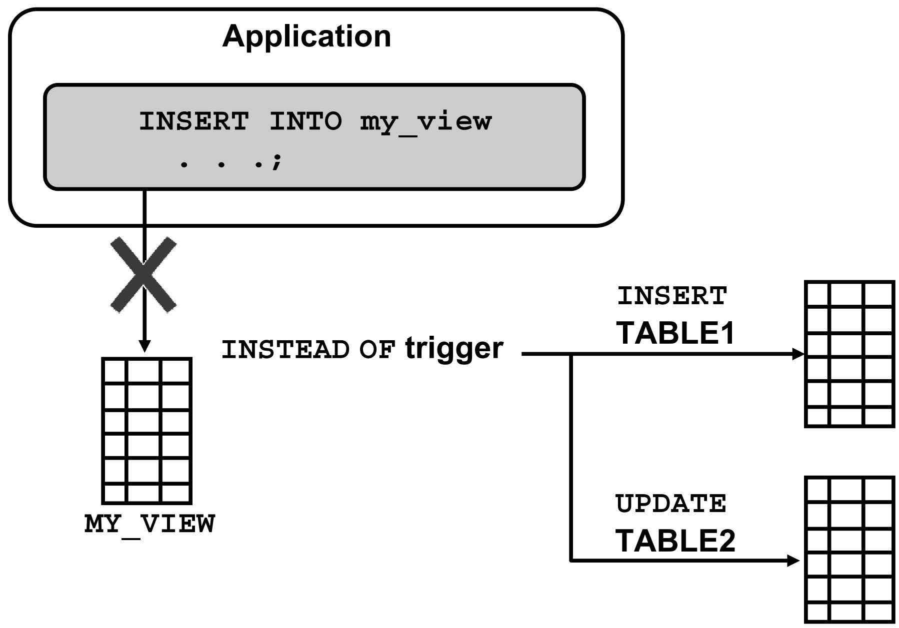
The example in the slide explains a situation in which the integrity constraint can be taken care of by using a trigger. The `EMPLOYEES` table has a foreign key constraint on the `DEPARTMENT_ID` column of the `DEPARTMENTS` table.

In the first SQL statement, the `DEPARTMENT_ID` of the employee 170 is modified to 999. Because department 999 does not exist in the `DEPARTMENTS` table, the statement raises exception –2292 for the integrity constraint violation.

The `EMPLOYEE_DEPT_FK_TRG` trigger is created that inserts a new row into the `DEPARTMENTS` table, using `:NEW.DEPARTMENT_ID` for the value of the new department's `DEPARTMENT_ID`. The trigger fires when the `UPDATE` statement modifies the `DEPARTMENT_ID` of employee 170 to 999. When the foreign key constraint is checked, it is successful because the trigger inserted the department 999 into the `DEPARTMENTS` table. Therefore, no exception occurs unless the department already exists when the trigger attempts to insert the new row. However, the `EXCEPTION` handler traps and masks the exception allowing the operation to succeed.

Note: This example works with Oracle8*i* and later releases but produces a run-time error in releases prior to Oracle8*i*.

INSTEAD OF Triggers



INSTEAD OF Triggers

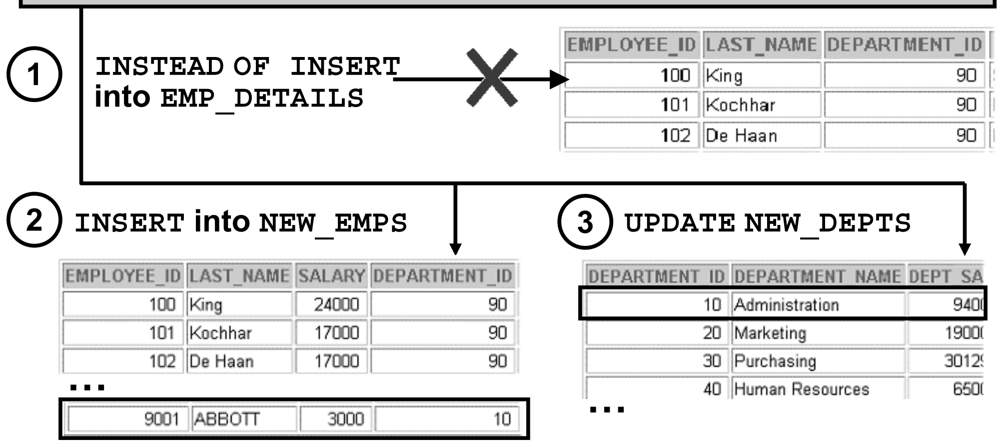
Use INSTEAD OF triggers to modify data in which the DML statement has been issued against an inherently nonupdatable view. These triggers are called INSTEAD OF triggers because, unlike other triggers, the Oracle server fires the trigger instead of executing the triggering statement. These triggers are used to perform INSERT, UPDATE, and DELETE operations directly on the underlying tables. You can write INSERT, UPDATE, and DELETE statements against a view, and the INSTEAD OF trigger works invisibly in the background to make the right actions take place. A view cannot be modified by normal DML statements if the view query contains set operators, group functions, clauses such as GROUP BY, CONNECT BY, START, the DISTINCT operator, or joins. For example, if a view consists of more than one table, an insert to the view may entail an insertion into one table and an update to another. So you write an INSTEAD OF trigger that fires when you write an insert against the view. Instead of the original insertion, the trigger body executes, which results in an insertion of data into one table and an update to another table.

Note: If a view is inherently updatable and has INSTEAD OF triggers, then the triggers take precedence. INSTEAD OF triggers are row triggers. The CHECK option for views is not enforced when insertions or updates to the view are performed by using INSTEAD OF triggers. The INSTEAD OF trigger body must enforce the check.

Creating an INSTEAD OF Trigger

Perform the INSERT into EMP_DETAILS that is based on EMPLOYEES and DEPARTMENTS tables:

```
INSERT INTO emp_details
VALUES (9001,'ABBOTT',3000, 10, 'Administration');
```



ORACLE

Creating an INSTEAD OF Trigger

You can create an INSTEAD OF trigger in order to maintain the base tables on which a view is based. The example illustrates an employee being inserted into view EMP_DETAILS, whose query is based on the EMPLOYEES and DEPARTMENTS tables. The NEW_EMP_DEPT (INSTEAD OF) trigger executes in place of the INSERT operation that causes the trigger to fire. The INSTEAD OF trigger then issues the appropriate INSERT and UPDATE to the base tables used by the EMP_DETAILS view. Therefore, instead of inserting the new employee record into the EMPLOYEES table, the following actions take place:

1. The NEW_EMP_DEPT INSTEAD OF trigger fires.
2. A row is inserted into the NEW_EMPS table.
3. The DEPT_SAL column of the NEW_DEPTS table is updated. The salary value supplied for the new employee is added to the existing total salary of the department to which the new employee has been assigned.

Note: The code for this scenario is shown in the next few pages.

Creating an INSTEAD OF Trigger

Use INSTEAD OF to perform DML on complex views:

```
CREATE TABLE new_emps AS
 SELECT employee_id,last_name,salary,department_id
 FROM employees;

CREATE TABLE new_depts AS
 SELECT d.department_id,d.department_name,
        sum(e.salary) dept_sal
 FROM employees e, departments d
 WHERE e.department_id = d.department_id;

CREATE VIEW emp_details AS
 SELECT e.employee_id, e.last_name, e.salary,
        e.department_id, d.department_name
 FROM employees e, departments d
 WHERE e.department_id = d.department_id
GROUP BY d.department_id,d.department_name;
```


Creating an INSTEAD OF Trigger (continued)

The example creates two new tables, NEW_EMPS and NEW_DEPTS, based on the EMPLOYEES and DEPARTMENTS tables, respectively. It also creates an EMP_DETAILS view from the EMPLOYEES and DEPARTMENTS tables.

If a view has a complex query structure, then it is not always possible to perform DML directly on the view to affect the underlying tables. The example requires creation of an INSTEAD OF trigger, called NEW_EMP_DEPT, shown on the next page. The NEW_DEPT_EMP trigger handles DML in the following way:

- When a row is inserted into the EMP_DETAILS view, instead of inserting the row directly into the view, rows are added into the NEW_EMPS and NEW_DEPTS tables, using the data values supplied with the INSERT statement.
- When a row is modified or deleted through the EMP_DETAILS view, corresponding rows in the NEW_EMPS and NEW_DEPTS tables are affected.

Note: INSTEAD OF triggers can be written only for views, and the BEFORE and AFTER timing options are not valid.

Creating an `INSTEAD OF` Trigger (continued)

```
CREATE OR REPLACE TRIGGER new_emp_dept
INSTEAD OF INSERT OR UPDATE OR DELETE ON emp_details
FOR EACH ROW
BEGIN
  IF INSERTING THEN
    INSERT INTO new_emps
    VALUES (:NEW.employee_id, :NEW.last_name,
            :NEW.salary, :NEW.department_id);
    UPDATE new_depts
      SET dept_sal = dept_sal + :NEW.salary
      WHERE department_id = :NEW.department_id;
  ELSIF DELETING THEN
    DELETE FROM new_emps
      WHERE employee_id = :OLD.employee_id;
    UPDATE new_depts
      SET dept_sal = dept_sal - :OLD.salary
      WHERE department_id = :OLD.department_id;
  ELSIF UPDATING ('salary') THEN
    UPDATE new_emps
      SET salary = :NEW.salary
      WHERE employee_id = :OLD.employee_id;
    UPDATE new_depts
      SET dept_sal = dept_sal +
                     (:NEW.salary - :OLD.salary)
      WHERE department_id = :OLD.department_id;
  ELSIF UPDATING ('department_id') THEN
    UPDATE new_emps
      SET department_id = :NEW.department_id
      WHERE employee_id = :OLD.employee_id;
    UPDATE new_depts
      SET dept_sal = dept_sal - :OLD.salary
      WHERE department_id = :OLD.department_id;
    UPDATE new_depts
      SET dept_sal = dept_sal + :NEW.salary
      WHERE department_id = :NEW.department_id;
  END IF;
END;
/
```

Comparison of Database Triggers and Stored Procedures

Triggers	Procedures
Defined with `CREATE TRIGGER`	Defined with `CREATE PROCEDURE`
Data dictionary contains source code in `USER_TRIGGERS`.	Data dictionary contains source code in `USER_SOURCE`.
Implicitly invoked by DML	Explicitly invoked
`COMMIT`, `SAVEPOINT`, and `ROLLBACK` are not allowed.	`COMMIT`, `SAVEPOINT`, and `ROLLBACK` are allowed.

Comparison of Database Triggers and Stored Procedures

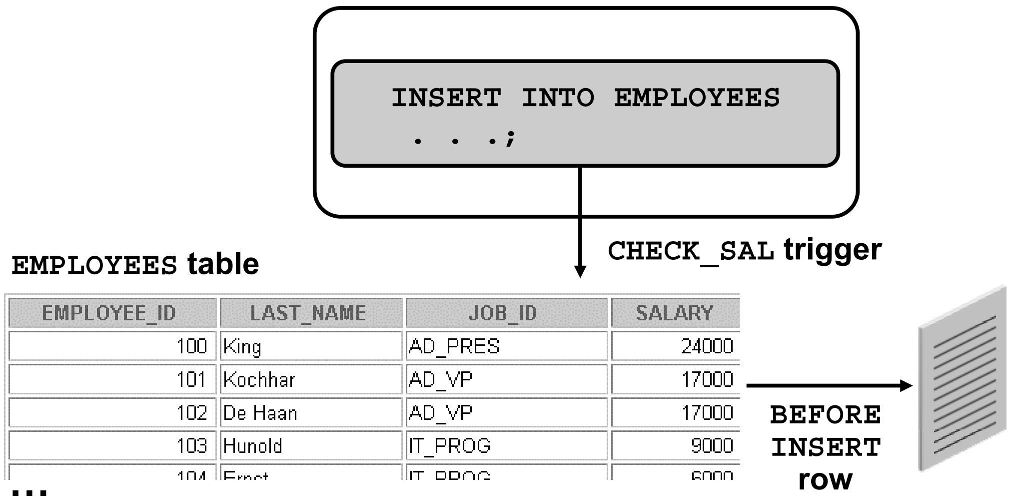
There are differences between database triggers and stored procedures:

Database Trigger	**Stored Procedure**
Invoked implicitly	Invoked explicitly
`COMMIT`, `ROLLBACK`, and `SAVEPOINT` statements are not allowed within the trigger body. It is possible to commit or roll back indirectly by calling a procedure, but it is not recommended because of side effects to transactions.	`COMMIT`, `ROLLBACK`, and `SAVEPOINT` statements are permitted within the procedure body.

Triggers are fully compiled when the `CREATE TRIGGER` command is issued and the executable code is stored in the data dictionary.

Note: If errors occur during the compilation of a trigger, the trigger is still created.

Comparison of Database Triggers and Oracle Forms Triggers



Comparison of Database Triggers and Oracle Forms Triggers

Database triggers are different from Forms Builder triggers.

Database Trigger	Forms Builder Trigger
Executed by actions from any database tool or application	Executed only within a particular Forms Builder application
Always triggered by a SQL DML, DDL, or a certain database action	Can be triggered by navigating from field to field, by pressing a key, or by many other actions
Is distinguished as either a statement or row trigger	Is distinguished as a statement or row trigger
Upon failure, causes the triggering statement to roll back	Upon failure, causes the cursor to freeze and may cause the entire transaction to roll back
Fires independently of, and in addition to, Forms Builder triggers	Fires independently of, and in addition to, database triggers
Executes under the security domain of the author of the trigger	Executes under the security domain of the Forms Builder user

Managing Triggers

- **Disable or reenable a database trigger:**

```
ALTER TRIGGER trigger_name DISABLE | ENABLE
```

- **Disable or reenable all triggers for a table:**

```
ALTER TABLE table_name DISABLE | ENABLE
  ALL TRIGGERS
```

- **Recompile a trigger for a table:**

```
ALTER TRIGGER trigger_name COMPILE
```


Managing Triggers

A trigger has two modes or states: `ENABLED` and `DISABLED`. When a trigger is first created, it is enabled by default. The Oracle server checks integrity constraints for enabled triggers and guarantees that triggers cannot compromise them. In addition, the Oracle server provides read-consistent views for queries and constraints, manages the dependencies, and provides a two-phase commit process if a trigger updates remote tables in a distributed database.

Disabling a Trigger

- By using the `ALTER TRIGGER` syntax, or disable all triggers on a table by using the `ALTER TABLE` syntax
- To improve performance or to avoid data integrity checks when loading massive amounts of data with utilities such as SQL*Loader. Consider disabling a trigger when it references a database object that is currently unavailable, due to a failed network connection, disk crash, offline data file, or offline tablespace.

Recompiling a Trigger

- By using the `ALTER TRIGGER` command to explicitly recompile a trigger that is invalid
- By issuing an `ALTER TRIGGER` statement with the `COMPILE` option, regardless of whether it is valid or invalid

Removing Triggers

To remove a trigger from the database, use the `DROP TRIGGER` statement:

```
DROP TRIGGER trigger_name;
```

Example:

```
DROP TRIGGER secure_emp;
```

Note: All triggers on a table are removed when the table is removed.

Removing Triggers

When a trigger is no longer required, use a SQL statement in *i*SQL*Plus to remove it.

Testing Triggers

- Test each triggering data operation, as well as nontriggering data operations.
- Test each case of the `WHEN` clause.
- Cause the trigger to fire directly from a basic data operation, as well as indirectly from a procedure.
- Test the effect of the trigger on other triggers.
- Test the effect of other triggers on the trigger.

ORACLE

Testing Triggers

Testing code can be a time-consuming process. Do the following when testing triggers:

- Ensure that the trigger works properly by testing a number of cases separately:
  - Test the most common success scenarios first.
  - Test the most common failure conditions to see that they are properly managed.
- The more complex the trigger, the more detailed your testing is likely to be. For example, if you have a row trigger with a `WHEN` clause specified, then you should ensure that the trigger fires when the conditions are satisfied. Or, if you have cascading triggers, you need to test the effect of one trigger on the other and ensure that you end up with the desired results.
- Use the `DBMS_OUTPUT` package to debug triggers.

Summary

In this lesson, you should have learned how to:

- **Create database triggers that are invoked by DML operations**
- **Create statement and row trigger types**
- **Use database trigger-firing rules**
- **Enable, disable, and manage database triggers**
- **Develop a strategy for testing triggers**
- **Remove database triggers**

ORACLE

Summary

This lesson covered creating database triggers that execute before, after, or instead of a specified DML operation. Triggers are associated with database tables or views. The `BEFORE` and `AFTER` timings apply to DML operations on tables. The `INSTEAD OF` trigger is used as a way to replace DML operations on a view with appropriate DML statements against other tables in the database.

Triggers are enabled by default but can be disabled to suppress their operation until enabled again. If business rules change, triggers can be removed or altered as required.

Practice 10: Overview

This practice covers the following topics:

- **Creating row triggers**
- **Creating a statement trigger**
- **Calling procedures from a trigger**

Practice 10: Overview

You create statement and row triggers in this practice. You create procedures that are invoked from the triggers.

Practice 10

1. The rows in the JOBS table store a minimum and maximum salary allowed for different JOB_ID values. You are asked to write code to ensure that employees' salaries fall in the range allowed for their job type, for insert and update operations.
   a. Write a procedure called CHECK_SALARY that accepts two parameters, one for an employee's job ID string and the other for the salary. The procedure uses the job ID to determine the minimum and maximum salary for the specified job. If the salary parameter does not fall within the salary range of the job, inclusive of the minimum and maximum, then it should raise an application exception, with the message "Invalid salary <sal>. Salaries for job <jobid> must be between <min> and <max>". Replace the various items in the message with values supplied by parameters and variables populated by queries. Save the file.
   b. Create a trigger called CHECK_SALARY_TRG on the EMPLOYEES table that fires before an INSERT or UPDATE operation on each row. The trigger must call the CHECK_SALARY procedure to carry out the business logic. The trigger should pass the new job ID and salary to the procedure parameters.
2. Test the CHECK_SAL_TRG using the following cases:
   a. Using your EMP_PKG.ADD_EMPLOYEE procedure, add employee Eleanor Beh to department 30. What happens and why?
   b. Update the salary of employee 115 to $2,000. In a separate update operation, change the employee job ID to HR_REP. What happens in each case?
   c. Update the salary of employee 115 to $2,800. What happens?
3. Update the CHECK_SALARY_TRG trigger to fire only when the job ID or salary values have actually changed.
   a. Implement the business rule using a WHEN clause to check whether the JOB_ID or SALARY values have changed.
   Note: Make sure that the condition handles the NULL in the OLD.column_name values if an INSERT operation is performed; otherwise, an insert operation will fail.
   b. Test the trigger by executing the EMP_PKG.ADD_EMPLOYEE procedure with the following parameter values: first_name='Eleanor', last name='Beh', email='EBEH', job='IT_PROG', sal=5000.
   c. Update employees with the IT_PROG job by incrementing their salary by $2,000. What happens?
   d. Update the salary to $9,000 for Eleanor Beh.
   Hint: Use an UPDATE statement with a subquery in the WHERE clause. What happens?
   e. Change the job of Eleanor Beh to ST_MAN using another UPDATE statement with a subquery. What happens?
4. You are asked to prevent employees from being deleted during business hours.
   a. Write a statement trigger called DELETE_EMP_TRG on the EMPLOYEES table to prevent rows from being deleted during weekday business hours, which are from 9:00 a.m. to 6:00 p.m.
   b. Attempt to delete employees with JOB_ID of SA_REP who are not assigned to a department.
   Hint: This is employee Grant with ID 178.

11

Applications for Triggers

Objectives

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

- Create additional database triggers
- Explain the rules governing triggers
- Implement triggers

Lesson Aim

In this lesson, you learn how to create more database triggers and learn the rules governing triggers. You also learn about the many applications of triggers.

Creating Database Triggers

- **Triggering a user event:**
  - `CREATE`, `ALTER`, or `DROP`
  - **Logging on or off**
- **Triggering database or system event:**
  - **Shutting down or starting up the database**
  - **A specific error (or any error) being raised**

Creating Database Triggers

Before coding the trigger body, decide on the components of the trigger.

Triggers on system events can be defined at the database or schema level. For example, a database shutdown trigger is defined at the database level. Triggers on data definition language (DDL) statements, or a user logging on or off, can also be defined at either the database level or schema level. Triggers on data manipulation language (DML) statements are defined on a specific table or a view.

A trigger defined at the database level fires for all users, and a trigger defined at the schema or table level fires only when the triggering event involves that schema or table.

Triggering events that can cause a trigger to fire:

- A data definition statement on an object in the database or schema
- A specific user (or any user) logging on or off
- A database shutdown or startup
- Any error that occurs

Creating Triggers on DDL Statements

Syntax:

```
CREATE [OR REPLACE] TRIGGER trigger_name
Timing
[ddl_event1 [OR ddl_event2 OR ...]]
ON {DATABASE|SCHEMA}
trigger_body
```


Creating Triggers on DDL Statements

DDL_Event	Possible Values
`CREATE`	Causes the Oracle server to fire the trigger whenever a `CREATE` statement adds a new database object to the dictionary
`ALTER`	Causes the Oracle server to fire the trigger whenever an `ALTER` statement modifies a database object in the data dictionary
`DROP`	Causes the Oracle server to fire the trigger whenever a `DROP` statement removes a database object in the data dictionary

The trigger body represents a complete PL/SQL block.

You can create triggers for these events on `DATABASE` or `SCHEMA`. You also specify `BEFORE` or `AFTER` for the timing of the trigger.

DDL triggers fire only if the object being created is a cluster, function, index, package, procedure, role, sequence, synonym, table, tablespace, trigger, type, view, or user.

Creating Triggers on System Events

Syntax:

```
CREATE [OR REPLACE] TRIGGER trigger_name
timing
[database_event1 [OR database_event2 OR ...]]
ON {DATABASE|SCHEMA}
trigger_body
```


Create Trigger Syntax

Database_event	Possible Values
AFTER SERVERERROR	Causes the Oracle server to fire the trigger whenever a server error message is logged
AFTER LOGON	Causes the Oracle server to fire the trigger whenever a user logs on to the database
BEFORE LOGOFF	Causes the Oracle server to fire the trigger whenever a user logs off the database
AFTER STARTUP	Causes the Oracle server to fire the trigger whenever the database is opened
BEFORE SHUTDOWN	Causes the Oracle server to fire the trigger whenever the database is shut down

You can create triggers for these events on DATABASE or SCHEMA, except SHUTDOWN and STARTUP, which apply only to DATABASE.

LOGON and LOGOFF Triggers: Example

```
CREATE OR REPLACE TRIGGER logon_trig
AFTER LOGON  ON  SCHEMA
BEGIN
 INSERT INTO log_trig_table(user_id,log_date,action)
 VALUES (USER, SYSDATE, 'Logging on');
END;
/
```

```
CREATE OR REPLACE TRIGGER logoff_trig
BEFORE LOGOFF  ON  SCHEMA
BEGIN
 INSERT INTO log_trig_table(user_id,log_date,action)
 VALUES (USER, SYSDATE, 'Logging off');
END;
/
```


LOGON and LOGOFF Triggers: Example

You can create these triggers to monitor how often you log on and off, or you may want to write a report that monitors the length of time for which you are logged on. When you specify `ON SCHEMA`, the trigger fires for the specific user. If you specify `ON DATABASE`, the trigger fires for all users.

CALL Statements

```
CREATE [OR REPLACE] TRIGGER trigger_name
timing
event1 [OR event2 OR event3]
ON table_name
[REFERENCING OLD AS old | NEW AS new]
[FOR EACH ROW]
[WHEN condition]
CALL procedure_name
/
```

```
CREATE OR REPLACE TRIGGER log_employee
BEFORE INSERT ON EMPLOYEES
CALL log_execution
/
```

Note: There is no semicolon at the end of the CALL statement.

CALL Statements

A CALL statement enables you to call a stored procedure, rather than code the PL/SQL body in the trigger itself. The procedure can be implemented in PL/SQL, C, or Java.

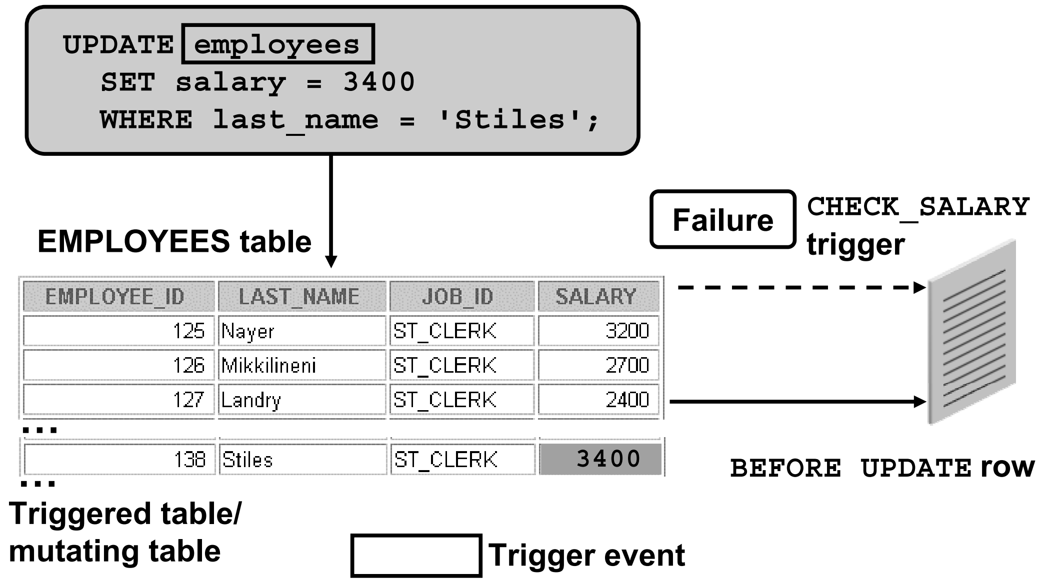
The call can reference the trigger attributes :NEW and :OLD as parameters, as in the following example:

```
CREATE TRIGGER salary_check
     BEFORE UPDATE OF salary, job_id ON employees
     FOR EACH ROW
     WHEN (NEW.job_id <> 'AD_PRES')
     CALL check_salary(:NEW.job_id, :NEW.salary)
     /
```

Note: There is no semicolon at the end of the CALL statement.

In the preceding example, the trigger calls a check_salary procedure. The procedure compares the new salary with the salary range for the new job ID from the JOBS table.

Reading Data from a Mutating Table



Rules Governing Triggers

Reading and writing data using triggers is subject to certain rules. The restrictions apply only to row triggers, unless a statement trigger is fired as a result of `ON DELETE CASCADE`.

Mutating Table

A mutating table is a table that is currently being modified by an `UPDATE`, `DELETE`, or `INSERT` statement, or a table that might need to be updated by the effects of a declarative `DELETE CASCADE` referential integrity action. For `STATEMENT` triggers, a table is not considered a mutating table.

The triggered table itself is a mutating table, as well as any table referencing it with the `FOREIGN KEY` constraint. This restriction prevents a row trigger from seeing an inconsistent set of data.

Mutating Table: Example

```
CREATE OR REPLACE TRIGGER check_salary
  BEFORE INSERT OR UPDATE OF salary, job_id
  ON employees
  FOR EACH ROW
  WHEN (NEW.job_id <> 'AD_PRES')
DECLARE
  minsalary employees.salary%TYPE;
  maxsalary employees.salary%TYPE;
BEGIN
  SELECT MIN(salary), MAX(salary)
   INTO  minsalary, maxsalary
   FROM  employees
   WHERE job_id = :NEW.job_id;
  IF :NEW.salary < minsalary OR
     :NEW.salary > maxsalary THEN
     RAISE_APPLICATION_ERROR(-20505,'Out of range');
  END IF;
END;
/
```


Mutating Table: Example

The CHECK_SALARY trigger in the example attempts to guarantee that whenever a new employee is added to the EMPLOYEES table or whenever an existing employee's salary or job ID is changed, the employee's salary falls within the established salary range for the employee's job.

When an employee record is updated, the CHECK_SALARY trigger is fired for each row that is updated. The trigger code queries the same table that is being updated. Therefore, it is said that the EMPLOYEES table is a mutating table.

Mutating Table: Example

```
UPDATE employees
 SET salary = 3400
 WHERE last_name = 'Stiles';
```

```
UPDATE employees
       *
ERROR at line 1:
ORA-04091: table PLSQL.EMPLOYEES is mutating, trigger/function may not see it
ORA-06512: at "PLSQL.CHECK_SALARY", line 5
ORA-04088: error during execution of trigger 'PLSQL.CHECK_SALARY'
```


Mutating Table: Example (continued)

In the example, the trigger code tries to read or select data from a mutating table.

If you restrict the salary within a range between the minimum existing value and the maximum existing value, then you get a run-time error. The `EMPLOYEES` table is mutating, or in a state of change; therefore, the trigger cannot read from it.

Remember that functions can also cause a mutating table error when they are invoked in a DML statement.

Possible Solutions

Possible solutions to this mutating table problem include the following:

- Store the summary data (the minimum salaries and the maximum salaries) in another summary table, which is kept up-to-date with other DML triggers.
- Store the summary data in a PL/SQL package, and access the data from the package. This can be done in a `BEFORE` statement trigger.

Depending on the nature of the problem, a solution can become more convoluted and difficult to solve. In this case, consider implementing the rules in the application or middle tier and avoid using database triggers to perform overly complex business rules.

Benefits of Database Triggers

- Improved data security:
  - Provide enhanced and complex security checks
  - Provide enhanced and complex auditing
- Improved data integrity:
  - Enforce dynamic data integrity constraints
  - Enforce complex referential integrity constraints
  - Ensure that related operations are performed together implicitly

ORACLE

Benefits of Database Triggers

You can use database triggers:

- As alternatives to features provided by the Oracle server
- If your requirements are more complex or more simple than those provided by the Oracle server
- If your requirements are not provided by the Oracle server at all

Managing Triggers

The following system privileges are required to manage triggers:

- The CREATE/ALTER/DROP (ANY) TRIGGER privilege that enables you to create a trigger in any schema
- The ADMINISTER DATABASE TRIGGER privilege that enables you to create a trigger on DATABASE
- The EXECUTE privilege (if your trigger refers to any objects that are not in your schema)

Note: Statements in the trigger body use the privileges of the trigger owner, not the privileges of the user executing the operation that fires the trigger.

Managing Triggers

To create a trigger in your schema, you need the CREATE TRIGGER system privilege, and you must either own the table specified in the triggering statement, have the ALTER privilege for the table in the triggering statement, or have the ALTER ANY TABLE system privilege. You can alter or drop your triggers without any further privileges being required.

If the ANY keyword is used, then you can create, alter, or drop your own triggers and those in another schema and can be associated with any user's table.

You do not need any privileges to invoke a trigger in your schema. A trigger is invoked by DML statements that you issue. But if your trigger refers to any objects that are not in your schema, the user creating the trigger must have the EXECUTE privilege on the referenced procedures, functions, or packages, and not through roles. As with stored procedures, statements in the trigger body use the privileges of the trigger owner, not the privileges of the user executing the operation that fires the trigger.

To create a trigger on DATABASE, you must have the ADMINISTER DATABASE TRIGGER privilege. If this privilege is later revoked, then you can drop the trigger but you cannot alter it.

Business Application Scenarios for Implementing Triggers

You can use triggers for:

- Security
- Auditing
- Data integrity
- Referential integrity
- Table replication
- Computing derived data automatically
- Event logging

Note: Appendix C covers each of these examples in more detail.

ORACLE

Business Application Scenarios for Implementing Triggers

Develop database triggers in order to enhance features that cannot otherwise be implemented by the Oracle server or as alternatives to those provided by the Oracle server.

Feature	Enhancement
Security	The Oracle server allows table access to users or roles. Triggers allow table access according to data values.
Auditing	The Oracle server tracks data operations on tables. Triggers track values for data operations on tables.
Data integrity	The Oracle server enforces integrity constraints. Triggers implement complex integrity rules.
Referential integrity	The Oracle server enforces standard referential integrity rules. Triggers implement nonstandard functionality.
Table replication	The Oracle server copies tables asynchronously into snapshots. Triggers copy tables synchronously into replicas.
Derived data	The Oracle server computes derived data values manually. Triggers compute derived data values automatically.
Event logging	The Oracle server logs events explicitly. Triggers log events transparently.

Viewing Trigger Information

You can view the following trigger information:

- `USER_OBJECTS` **data dictionary view: Object information**
- `USER_TRIGGERS` **data dictionary view: Text of the trigger**
- `USER_ERRORS` **data dictionary view: PL/SQL syntax errors (compilation errors) of the trigger**

ORACLE

Viewing Trigger Information

The slide shows the data dictionary views that you can access to get information regarding the triggers.

The `USER_OBJECTS` view contains the name and status of the trigger and the date and time when the trigger was created.

The `USER_ERRORS` view contains the details about the compilation errors that occurred while a trigger was compiling. The contents of these views are similar to those for subprograms.

The `USER_TRIGGERS` view contains details such as name, type, triggering event, the table on which the trigger is created, and the body of the trigger.

The `SELECT Username FROM USER_USERS;` statement gives the name of the owner of the trigger, not the name of the user who is updating the table.

Using USER_TRIGGERS

Column	Column Description
TRIGGER_NAME	Name of the trigger
TRIGGER_TYPE	The type is BEFORE, AFTER, INSTEAD OF
TRIGGERING_EVENT	The DML operation firing the trigger
TABLE_NAME	Name of the database table
REFERENCING_NAMES	Name used for :OLD and :NEW
WHEN_CLAUSE	The when_clause used
STATUS	The status of the trigger
TRIGGER_BODY	The action to take

* **Abridged column list**

Using USER_TRIGGERS

If the source file is unavailable, then you can use *i*SQL*Plus to regenerate it from USER_TRIGGERS. You can also examine the ALL_TRIGGERS and DBA_TRIGGERS views, each of which contains the additional column OWNER, for the owner of the object.

Listing the Code of Triggers

```
SELECT trigger_name, trigger_type, triggering_event,
       table_name, referencing_names,
       status, trigger_body
FROM   user_triggers
WHERE  trigger_name = 'RESTRICT_SALARY';
```

TRIGGER_NAME	TRIGGER_TYPE	TRIGGERING_EVENT	TABLE_NAME	REFERENCING_NAMES	WHEN_CLAUS	STATUS	TRIGGER_BODY
RESTRICT_SALARY	BEFORE EACH ROW	INSERT OR UPDATE	EMPLOYEES	REFERENCING NEW AS NEW OLD AS OLD		ENABLED	BEGIN IF NOT (:NEW.JOB_ ID IN ('AD_PRES ', 'AD_VP')) AND :NE W.SAL

Example

Use the USER_TRIGGERS data dictionary view to display information about the RESTRICT_SALARY trigger.

Summary

In this lesson, you should have learned how to:

- Use advanced database triggers
- List mutating and constraining rules for triggers
- Describe real-world applications of triggers
- Manage triggers
- View trigger information

Practice 11: Overview

This practice covers the following topics:

- **Creating advanced triggers to manage data integrity rules**
- **Creating triggers that cause a mutating table exception**
- **Creating triggers that use package state to solve the mutating table problem**

Practice 11: Overview

In this practice, you implement a simple business rule for ensuring data integrity of employees' salaries with respect to the valid salary range for their job. You create a trigger for this rule.

During this process, your new triggers cause a cascading effect with triggers created in the practice section of the lesson titled "Creating Triggers." The cascading effect results in a mutating table exception on the `JOBS` table. You then create a PL/SQL package and additional triggers to solve the mutating table issue.

Practice 11

1. Employees receive an automatic increase in salary if the minimum salary for a job is increased to a value larger than their current salary. Implement this requirement through a package procedure called by a trigger on the JOBS table. When you attempt to update the minimum salary in the JOBS table and try to update the employees' salary, the CHECK_SALARY trigger attempts to read the JOBS table, which is subject to change, and you get a mutating table exception that is resolved by creating a new package and additional triggers.
   a. Update your EMP_PKG package (from Practice 7) by adding a procedure called SET_SALARY that updates the employees' salaries. The procedure accepts two parameters: the job ID for those salaries that may have to be updated, and the new minimum salary for the job ID. The procedure sets all the employees' salaries to the minimum for their jobs if their current salaries are less than the new minimum value.
   b. Create a row trigger named UPD_MINSALARY_TRG on the JOBS table that invokes the EMP_PKG.SET_SALARY procedure, when the minimum salary in the JOBS table is updated for a specified job ID.
   c. Write a query to display the employee ID, last name, job ID, current salary, and minimum salary for employees who are programmers—that is, their JOB_ID is 'IT_PROG'. Then update the minimum salary in the JOBS table to increase it by $1,000. What happens?
2. To resolve the mutating table issue, you create a JOBS_PKG to maintain in memory a copy of the rows in the JOBS table. Then the CHECK_SALARY procedure is modified to use the package data rather than issue a query on a table that is mutating to avoid the exception. However, a BEFORE INSERT OR UPDATE statement trigger must be created on the EMPLOYEES table to initialize the JOBS_PKG package state before the CHECK_SALARY row trigger is fired.
   a. Create a new package called JOBS_PKG with the following specification:
   ```
   PROCEDURE initialize;
   FUNCTION get_minsalary(jobid VARCHAR2) RETURN NUMBER;
   FUNCTION get_maxsalary(jobid VARCHAR2) RETURN NUMBER;
   PROCEDURE set_minsalary(jobid VARCHAR2,min_salary NUMBER);
   PROCEDURE set_maxsalary(jobid VARCHAR2,max_salary NUMBER);
   ```
   b. Implement the body of the JOBS_PKG, where:
   You declare a private PL/SQL index-by table called jobs_tabtype that is indexed by a string type based on the JOBS.JOB_ID%TYPE.
   You declare a private variable called jobstab based on the jobs_tabtype.
   The INITIALIZE procedure reads the rows in the JOBS table by using a cursor loop, and uses the JOB_ID value for the jobstab index that is assigned its corresponding row. The GET_MINSALARY function uses a jobid parameter as an index to the jobstab and returns the min_salary for that element. The GET_MAXSALARY function uses a jobid parameter as an index to the jobstab and returns the max_salary for that element.

The `SET_MINSALARY` procedure uses its `jobid` as an index to the `jobstab` to set the `min_salary` field of its element to the value in the `min_salary` parameter.
The `SET_MAXSALARY` procedure uses its `jobid` as an index to the `jobstab` to set the `max_salary` field of its element to the value in the `max_salary` parameter.

c. Copy the `CHECK_SALARY` procedure from Practice 10, Exercise 1a, and modify the code by replacing the query on the `JOBS` table with statements to set the local `minsal` and `maxsal` variables with values from the `JOBS_PKG` data by calling the appropriate `GET_*SALARY` functions. This step should eliminate the mutating trigger exception.

d. Implement a `BEFORE INSERT OR UPDATE` statement trigger called `INIT_JOBPKG_TRG` that uses the `CALL` syntax to invoke the `JOBS_PKG.INITIALIZE` procedure to ensure that the package state is current before the DML operations are performed.

e. Test the code changes by executing the query to display the employees who are programmers, then issue an update statement to increase the minimum salary of the `IT_PROG` job type by `1000` in the `JOBS` table, followed by a query on the employees with the `IT_PROG` job type to check the resulting changes. Which employees' salaries have been set to the minimum for their jobs?

3. Because the `CHECK_SALARY` procedure is fired by the `CHECK_SALARY_TRG` before inserting or updating an employee, you must check whether this still works as expected.

   a. Test this by adding a new employee using `EMP_PKG.ADD_EMPLOYEE` with the following parameters: `('Steve', 'Morse', 'SMORSE'`, and `sal => 6500)`. What happens?

   b. To correct the problem encountered when adding or updating an employee, create a `BEFORE INSERT OR UPDATE` statement trigger called `EMPLOYEE_INITJOBS_TRG` on the `EMPLOYEES` table that calls the `JOBS_PKG.INITIALIZE` procedure. Use the `CALL` syntax in the trigger body.

   c. Test the trigger by adding employee Steve Morse again. Confirm the inserted record in the `employees` table by displaying the employee ID, first and last names, salary, job ID, and department ID.

Understanding and Influencing the PL/SQL Compiler

12

ORACLE

Objectives

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

- **Describe native and interpreted compilations**
- **List the features of native compilation**
- **Switch between native and interpreted compilations**
- **Set parameters that influence PL/SQL compilation**
- **Query data dictionary views on how PL/SQL code is compiled**
- **Use the compiler warning mechanism and the `DBMS_WARNING` package to implement compiler warnings**

Lesson Aim

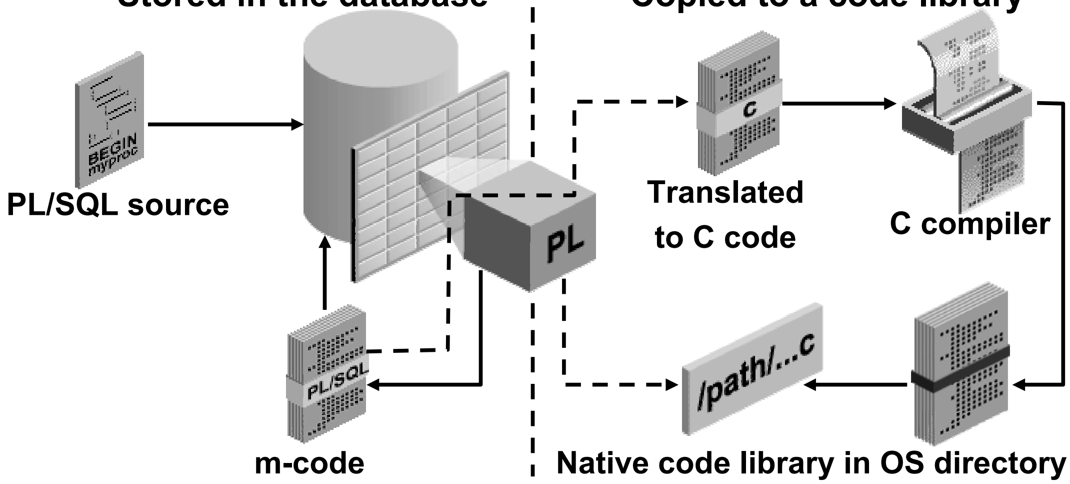
In this lesson, you learn to distinguish between native and interpreted compilation of PL/SQL code. The lesson discusses how to use native compilation, which is the default, for Oracle Database 10*g* with the benefit of having faster execution time for your PL/SQL code.

You also learn how to influence the compiler settings by setting variable session parameters, or using the programmatic interface provided by the `DBMS_WARNING` package. The lesson covers query compilation settings using the `USER_STORED_SETTINGS` and `USER_PLSQL_OBJECTS` data dictionary views.

Native and Interpreted Compilation

Interpreted code
- **Compiled to m-code**
- **Stored in the database**

Natively compiled code
- **Translated C and compiled**
- **Copied to a code library**



Native and Interpreted Compilation

As depicted in the slide, on the left of the vertical dotted line, a program unit processed as interpreted PL/SQL is compiled into machine-readable code (m-code), which is stored in the database and interpreted at run time.

On the right of the vertical dotted line, the PL/SQL source is subjected to native compilation, where the PL/SQL statements are compiled to m-code that is translated into C code. The m-code is not retained. The C code is compiled with the usual C compiler and linked to the Oracle process using native machine code library. The code library is stored in the database but copied to a specified directory path in the operating system, from which it is loaded at run time. Native code bypasses the typical run-time interpretation of code.

Note: Native compilation cannot do much to speed up SQL statements called from PL/SQL, but it is most effective for computation-intensive PL/SQL procedures that do not spend most of their time executing SQL.

You can natively compile both the supplied Oracle packages and your own PL/SQL code. Compiling all PL/SQL code in the database means that you see the speedup in your own code and all the built-in PL/SQL packages. If you decide that you will have significant performance gains in database operations using PL/SQL native compilation, Oracle recommends that you compile the whole database using the `NATIVE` setting.

Features and Benefits of Native Compilation

Native compilation:

- **Uses a generic `makefile` that uses the following operating system software:**
  - **C compiler**
  - **Linker**
  - **Make utility**
- **Generates shared libraries that are copied to the file system and loaded at run time**
- **Provides better performance (up to 30% faster than interpreted code) for computation-intensive procedural operations**

ORACLE

Features and Benefits of Native Compilation

The PL/SQL native compilation process makes use of a `makefile`, called `spnc_makefile.mk`, located in the `$ORACLE_HOME/plsql` directory. The `makefile` is processed by the Make utility that invokes the C compiler, which is the linker on the supported operating system, to compile and link the resulting C code into shared libraries. The shared libraries are stored inside the database and are copied to the file system. At run time, the shared libraries are loaded and run when the PL/SQL subprogram is invoked.

In accordance with Optimal Flexible Architecture (OFA) recommendations, the shared libraries should be stored near the data files. C code runs faster than PL/SQL, but it takes longer to compile than m-code. PL/SQL native compilation provides the greatest performance gains for computation-intensive procedural operations.

Examples of such operations are data warehouse applications and applications with extensive server-side transformations of data for display. In such cases, expect speed increases of up to 30%.

Considerations When Using Native Compilation

Consider the following:

- **Debugging tools for PL/SQL cannot debug natively compiled code.**
- **Natively compiled code is slower to compile than interpreted code.**
- **Large amounts of natively compiled subprograms can affect performance due to operating system–imposed limitations when handling shared libraries. OS directory limitations can be managed by setting database initialization parameters:**
  - `PLSQL_NATIVE_LIBRARY_SUBDIR_COUNT` **and**
  - `PLSQL_NATIVE_LIBRARY_DIR`

Limitations of Native Compilation

As stated, the key benefit of natively compiled code is faster execution, particularly for computationally intensive PL/SQL code, as much as 30% more. Consider that:

- Debugging tools for PL/SQL do not handle procedures compiled for native execution. Therefore, use interpreted compilation in development environments, and natively compile the code in a production environment.
- The compilation time increases when using native compilation because of the requirement to translate the PL/SQL statement to its C equivalent and execute the Make utility to invoke the C compiler and linker for generating the resulting compiled code library.
- If many procedures and packages (more than 5,000) are compiled for native execution, a large number of shared objects in a single directory may affect performance. The operating system directory limitations can be managed by automatically distributing libraries across several subdirectories. To do this, perform the following tasks before natively compiling the PL/SQL code:
  - Set the `PLSQL_NATIVE_LIBRARY_SUBDIR_COUNT` database initialization parameter to a large value, such as 1,000, before creating the database or compiling the PL/SQL packages or procedures.
  - Create `PLSQL_NATIVE_LIBRARY_SUBDIR_COUNT` subdirectories in the path specified in the `PLSQL_NATIVE_LIBRARY_DIR` initialization parameter.

Parameters Influencing Compilation

System parameters are set in the `initSID.ora` file or by using the `SPFILE`:

```
PLSQL_NATIVE_LIBRARY_DIR = full-directory-path-name
PLSQL_NATIVE_LIBRARY_SUBDIR_COUNT = count
```

System or session parameters

```
PLSQL_COMPILER_FLAGS = 'NATIVE' or 'INTERPRETED'
```


Parameters Influencing Compilation

In all circumstances, whether you intend to compile a database as `NATIVE` or you intend to compile individual PL/SQL units at the session level, you must set all required parameters.

The system parameters are set in the `initSID.ora` file by using the `SPFILE` mechanism. Two parameters that are set as system-level parameters are the following:

- The `PLSQL_NATIVE_LIBRARY_DIR` value, which specifies the full path and directory name used to store the shared libraries that contain natively compiled PL/SQL code
- The `PLSQL_NATIVE_LIBRARY_SUBDIR_COUNT` value, which specifies the number of subdirectories in the directory specified by the `PLSQL_NATIVE_LIBRARY_DIR` parameter. Use a script to create directories with consistent names (for example, `d0`, `d1`, `d2`, and so on), and then the libraries are automatically distributed among these subdirectories by the PL/SQL compiler.

By default, PL/SQL program units are kept in one directory.

The `PLSQL_COMPILER_FLAGS` parameter can be set to a value of `NATIVE` or `INTERPRETED`, either as a database initialization for a systemwide default or for each session using an `ALTER SESSION` statement.

Switching Between Native and Interpreted Compilation

- **Setting native compilation:**
  - **For the system:**

```
ALTER SYSTEM SET plsql_compiler_flags='NATIVE';
```

  - **For the session:**

```
ALTER SESSION SET plsql_compiler_flags='NATIVE';
```

- **Setting interpreted compilation:**
  - **For the system level:**

```
ALTER SYSTEM
        SET plsql_compiler_flags='INTERPRETED';
```

  - **For the session:**

```
ALTER SESSION
        SET plsql_compiler_flags='INTERPRETED';
```

ORACLE

Switching Between Native and Interpreted Compilation

The PLSQL_COMPILER_FLAGS parameter determines whether PL/SQL code is natively compiled or interpreted, and determines whether debug information is included. The default setting is INTERPRETED,NON_DEBUG. To enable PL/SQL native compilation, you must set the value of PLSQL_COMPILER_FLAGS to NATIVE.

If you compile the whole database as NATIVE, then Oracle recommends that you set PLSQL_COMPILER_FLAGS at the system level.

To set compilation type at the system level (usually done by a DBA), execute the following statements:

```
ALTER SYSTEM SET plsql_compiler_flags='NATIVE'
ALTER SYSTEM SET plsql_compiler_flags='INTERPRETED'
```

To set compilation type at the session level, execute one of the following statements:

```
ALTER SESSION SET plsql_compiler_flags='NATIVE'
ALTER SESSION SET plsql_compiler_flags='INTERPRETED'
```

Viewing Compilation Information in the Data Dictionary

Query information in the following views:

- `USER_STORED_SETTINGS`
- `USER_PLSQL_OBJECTS`

Example:

```
SELECT param_value
FROM   user_stored_settings
WHERE  param_name = 'plsql_compiler_flags'
  AND  object_name = 'GET_EMPLOYEES';
```

Note: The `PARAM_VALUE` column has a value of `NATIVE` for procedures that are compiled for native execution; otherwise, it has a value of `INTERPRETED`.

ORACLE

Viewing Compilation Information in the Data Dictionary

To check whether an existing procedure is compiled for native execution or not, you can query the following data dictionary views:

```
[USER | ALL | DBA]_STORED_SETTINGS
[USER | ALL | DBA ]_PLSQL_OBJECTS
```

The example in the slide shows how you can check the status of the procedure called `GET_EMPLOYEES`. The `PARAM_VALUE` column has a value of `NATIVE` for procedures that are compiled for native execution; otherwise, it has a value of `INTERPRETED`.

After procedures are natively compiled and turned into shared libraries, they are automatically linked into the Oracle process. You do not need to restart the database, or move the shared libraries to a different location. You can call back and forth between stored procedures, whether they are all compiled interpreted (the default), all compiled for native execution, or a mixture of both.

Because the `PLSQL_COMPILER_FLAGS` setting is stored inside the library unit for each procedure, the procedures compiled for native execution are compiled the same way when the procedure is recompiled automatically after being invalidated, such as when a table that it depends on is re-created.

Using Native Compilation

To enable native compilation, perform the following steps:

1. **Edit the supplied `makefile` and enter appropriate paths and other values for your system.**
2. **Set the `PLSQL_COMPILER_FLAGS` parameter (at system or session level) to the value `NATIVE`. The default is `INTERPRETED`.**
3. **Compile the procedures, functions, and packages.**
4. **Query the data dictionary to see that a procedure is compiled for native execution.**

ORACLE

Using Native Compilation

To enable native compilation, perform the following steps:

1. Check and edit the compiler, linker, utility paths, and other values, if required.
2. Set the `PLSQL_COMPILER_FLAGS` to `NATIVE`.
3. Compile the procedures, functions, and packages. Compiling can be done by:
   - Using the appropriate `ALTER PROCEDURE`, `ALTER FUNCTION`, or `ALTER PACKAGE` statements with the `COMPILE` option
   - Dropping the procedure and re-creating it
   - Running one of the SQL*Plus scripts that sets up a set of Oracle-supplied packages
   - Creating a database using a preconfigured initialization file with its `PLSQL_COMPILER_FLAGS` set to `NATIVE`
4. Confirm the compilation type using the appropriate data dictionary tables.

Note: Dependencies between database objects are handled in the same manner as in previous Oracle database versions. If an object on which a natively compiled PL/SQL program unit depends changes, then the PL/SQL module is invalidated. The next time the same program unit is executed, the RDBMS attempts to revalidate the module. When a module is recompiled as part of revalidation, it is compiled using the setting that was used the last time the module was compiled, and it is saved in the `*_STORED_SETTINGS` view.

Compiler Warning Infrastructure

The PL/SQL compiler in Oracle Database 10*g* has been enhanced to produce warnings for subprograms. Warning levels:

- **Can be set:**
  - **Declaratively with the `PLSQL_WARNINGS` initialization parameter**
  - **Programmatically using the `DBMS_WARNINGS` package**
- **Are arranged in three categories: severe, performance, and informational**
- **Can be enabled and disabled by category or a specific message**

Examples of warning messages:

SP2-0804: Procedure created with compilation warnings

PLW-07203: Parameter `'IO_TBL'` may benefit from use of the `NOCOPY` compiler hint.

Compiler Warning Infrastructure

The Oracle PL/SQL compiler can issue warnings when you compile subprograms that produce ambiguous results or use inefficient constructs. You can selectively enable and disable these warnings:

- Declaratively by setting the `PLSQL_WARNINGS` initialization parameter
- Programmatically using the `DBMS_WARNINGS` package

The warning level is arranged in the following categories: severe, performance, and informational. Warnings levels can be enabled or disabled by category or by a specific warning message number.

Benefits of Compiler Warnings

Using compiler warnings can help to:

- Make your programs more robust and avoid problems at run time
- Identify potential performance problems
- Indicate factors that produce undefined results

Note: You can enable checking for certain warning conditions when these conditions are not serious enough to produce an error and keep you from compiling a subprogram.

Setting Compiler Warning Levels

Set the `PLSQL_WARNINGS` initialization parameter to enable the database to issue warning messages.

```
ALTER SESSION SET PLSQL_WARNINGS = 'ENABLE:SEVERE',
                   'DISABLE:INFORMATIONAL';
```

- **The `PLSQL_WARNINGS` combine a qualifier value (`ENABLE`, `DISABLE`, or `ERROR`) with a comma-separated list of message numbers, or with one of the following modifier values:**
  - **`ALL`, `SEVERE`, `INFORMATIONAL`, or `PERFORMANCE`**
- **Warning messages use a `PLW` prefix.**

  PLW-07203: Parameter `'IO_TBL'` may benefit from use of the `NOCOPY` compiler hint.

Setting Compiler Warning Levels

The PLSQL_WARNINGS setting enables or disables the reporting of warning messages by the PL/SQL compiler, and specifies which warning messages to show as errors. The PLSQL_WARNINGS parameter can be set for the system using the initialization file or the ALTER SYSTEM statement, or for the session using the ALTER SESSION statement as shown in the example in the slide. By default, the value is set to DISABLE:ALL.

The parameter value comprises a comma-separated list of quoted qualifier and modifier keywords, where the keywords are separated by colons. The qualifier values are:

- **ENABLE:** To enable a specific warning or a set of warnings
- **DISABLE:** To disable a specific warning or a set of warnings
- **ERROR:** To treat a specific warning or a set of warnings as errors

The modifier value ALL applies to all warning messages. SEVERE, INFORMATIONAL, and PERFORMANCE apply to messages in their own category, and an integer list for specific warning messages. For example:

```
PLSQL_WARNINGS='ENABLE:SEVERE','DISABLE:INFORMATIONAL';
PLSQL_WARNINGS='DISABLE:ALL';
PLSQL_WARNINGS='DISABLE:5000','ENABLE:5001','ERROR:5002';
PLSQL_WARNINGS='ENABLE:(5000,5001)','DISABLE:(6000)';
```

Guidelines for Using PLSQL_WARNINGS

The PLSQL_WARNINGS setting:

- Can be set to DEFERRED at the system level
- Is stored with each compiled subprogram
- That is current for the session is used, by default, when recompiling with:
  - A CREATE OR REPLACE statement
  - An ALTER...COMPILE statement
- That is stored with the compiled subprogram is used when REUSE SETTINGS is specified when recompiling with an ALTER...COMPILE statement

ORACLE

Guidelines for Using PLSQL_WARNINGS

As already stated, the PLSQL_WARNINGS parameter can be set at the session level or the system level. When setting it at the system level, you can include the value DEFERRED so that it applies to future sessions but not the current one.

The settings for the PLSQL_WARNINGS parameter are stored along with each compiled subprogram. If you recompile the subprogram with a CREATE OR REPLACE statement, the current settings for that session are used. If you recompile the subprogram with an ALTER...COMPILE statement, then the current session setting is used unless you specify the REUSE SETTINGS clause in the statement, which uses the original setting that is stored with the subprogram.

DBMS_WARNING Package

The DBMS_WARNING package provides a way to programmatically manipulate the behavior of current system or session PL/SQL warning settings. Using DBMS_WARNING subprograms, you can:

- Query existing settings
- Modify the settings for specific requirements or restore original settings
- Delete the settings

Example: Saving and restoring warning settings for a development environment that calls your code that compiles PL/SQL subprograms, and suppresses warnings due to business requirements

DBMS_WARNING Package

The DBMS_WARNING package provides a way to manipulate the behavior of PL/SQL warning messages, in particular, by reading and changing the setting of the PLSQL_WARNINGS initialization parameter to control what kinds of warnings are suppressed, displayed, or treated as errors. This package provides the interface to query, modify, and delete current system or session settings.

The DBMS_WARNINGS package is valuable if you are writing a development environment that compiles PL/SQL subprograms. Using the package interface routines, you can control PL/SQL warning messages programmatically to suit your requirements.

Here is an example: Suppose you write some code to compile PL/SQL code. You know that the compiler will issue performance warnings when passing collection variables as OUT or IN OUT parameters without specifying the NOCOPY hint. The general environment that calls your compilation utility may or may not have appropriate warning level settings. In any case, your business rules indicate that the calling environment set must be preserved and that your compilation process should suppress the warnings. By calling subprograms in the DBMS_WARNINGS package, you can detect the current warning settings, change the setting to suit your business requirements, and restore the original settings when your processing has completed.

Using DBMS_WARNING Procedures

- **Package procedures change PL/SQL warnings:**

```
ADD_WARNING_SETTING_CAT(w_category,w_value,scope)
ADD_WARNING_SETTING_NUM(w_number,w_value,scope)
SET_WARNING_SETTING_STRING(w_value, scope)
```

  - **All parameters are `IN` parameters and have the `VARCHAR2` data type. However, the `w_number` parameter is a `NUMBER` data type.**
  - **Parameter string values are not case sensitive.**
  - **The `w_value` parameters values are `ENABLE`, `DISABLE`, and `ERROR`.**
  - **The `w_category` values are `ALL`, `INFORMATIONAL`, `SEVERE`, and `PERFORMANCE`.**
  - **The `scope` value is either `SESSION` or `SYSTEM`. Using `SYSTEM` requires the `ALTER SYSTEM` privilege.**

ORACLE

Using DBMS_WARNING Procedures

The package procedures are the following:

- **ADD_WARNING_SETTING_CAT:** Modifies the current session or system warning settings of the `warning_category` previously supplied
- **ADD_WARNING_SETTING_NUM:** Modifies the current session or system warning settings of the `warning_number` previously supplied
- **SET_WARNING_SETTING_STRING:** Replaces previous settings with the new value

Using the `SET_WARNING_SETTING_STRING`, you can set one warning setting. If you have multiple warning settings, you should perform the following steps:

1. Call `SET_WARNING_SETTING_STRING` to set the initial warning setting string.
2. Call `ADD_WARNING_SETTING_CAT` (or `ADD_WARNING_SETTING_NUM`) repeatedly to add additional settings to the initial string.

Here is an example to establish the following warning setting string in the current session:

```
ENABLE:INFORMATIONAL,DISABLE:PERFORMANCE,ENABLE:SEVERE
```

Execute the following two lines of code:

```
dbms_warning.set_warning_setting_string('ENABLE:ALL','session');
dbms_warning.add_warning_setting_cat('PERFORMANCE','disable',
                                     'session');
```

Using `DBMS_WARNING` Functions

- **Package functions read PL/SQL warnings:**

```
GET_CATEGORY(w_number) RETURN VARCHAR2
GET_WARNING_SETTING_CAT(w_category)RETURN VARCHAR2
GET_WARNING_SETTING_NUM(w_number) RETURN VARCHAR2
GET_WARNING_SETTING_STRING RETURN VARCHAR2
```

  - **`GET_CATEGORY` returns a value of `ALL`, `INFORMATIONAL`, `SEVERE`, or `PERFORMANCE` for a given message number.**
  - **`GET_WARNING_SETTING_CAT` returns `ENABLE`, `DISABLE`, or `ERROR` as the current warning value for a category name, and `GET_WARNING_SETTING_NUM` returns the value for a specific message number.**
  - **`GET_WARNING_SETTING_STRING` returns the entire warning string for the current session.**

Using DBMS_WARNING Functions

The following is a list of package functions:

- GET_CATEGORY returns the category name for the given message number.
- GET_WARNING_SETTING_CAT returns the current session warning setting for the specified category.
- GET_WARNING_SETTING_NUM returns the current session warning setting for the specified message number.
- GET_WARNING_SETTING_STRING returns the entire warning string for the current session.

To determine the current session warning settings, enter:

```
EXECUTE DBMS_OUTPUT.PUT_LINE( -
      DBMS_WARNING.GET_WARNING_SETTING_STRING);
```

To determine the category for warning message number PLW-07203, use:

```
EXECUTE DBMS_OUTPUT.PUT_LINE( -
      DBMS_WARNING.GET_CATEGORY(7203))
```

The result string should be PERFORMANCE.

Note: The message numbers must be specified as positive integers because the data type for the GET_CATEGORY parameter is PLS_INTEGER (allowing positive integer values).

Using DBMS_WARNING: Example

Consider the following scenario:
Save current warning settings, disable warnings for the PERFORMANCE category, compile a PL/SQL package, and restore the original warning setting.

```
CREATE PROCEDURE compile(pkg_name VARCHAR2) IS
  warn_value VARCHAR2(200);
  compile_stmt VARCHAR2(200) :=
    'ALTER PACKAGE '|| pkg_name ||' COMPILE';
BEGIN
  warn_value :=   -- Save current settings
      DBMS_WARNING.GET_WARNING_SETTING_STRING;
  DBMS_WARNING.ADD_WARNING_SETTING_CAT( -- change
      'PERFORMANCE', 'DISABLE', 'SESSION');
  EXECUTE IMMEDIATE compile_stmt;
  DBMS_WARNING.SET_WARNING_SETTING_STRING(--restore
      warn_value, 'SESSION');
END;
```


Using DBMS_WARNING: Example

In the slide, the example of the compile procedure is designed to compile a named PL/SQL package. The business rules require the following:

- Warnings in the performance category are suppressed.
- The calling environment's warning settings must be restored after the compilation is performed.

The code does not know or care about what the calling environment warning settings are; it simply uses the DBMS_WARNING.GET_WARNING_SETTING_STRING function to save the current setting.

This value is used to restore the calling environment setting using the DBMS_WARNING.SET_WARNING_SETTING_STRING procedure in the last line of the example code. Before compiling the package using Native Dynamic SQL, the compile procedure alters the current session warning level by disabling warnings for the PERFORMANCE category.

For example, the compiler will suppress warnings about PL/SQL parameters passed using OUT or IN OUT modes that do not specify the NOCOPY hint to gain better performance.

Using DBMS_WARNING: Example

To test the compile procedure, you can use the following script sequence in *i*SQL*Plus:

```
DECLARE
  PROCEDURE print(s VARCHAR2) IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE(s);
  END;
BEGIN
  print('Warning settings before: '||
         DBMS_WARNING.GET_WARNING_SETTING_STRING);
  compile('my_package');
  print('Warning settings after: '||
         DBMS_WARNING.GET_WARNING_SETTING_STRING);
END;
/
SHOW ERRORS PACKAGE MY_PACKAGE
```

ORACLE

Using DBMS_WARNING: Example (continued)

The slide shows an anonymous block that is used to display the current warning settings for the session before compilation takes place, executes the compile procedure, and prints the current warning settings for the session again. The before and after values for the warning settings should be identical.

The last line containing the SHOW ERRORS PACKAGE MY_PACKAGE is used to verify whether the warning messages in the performance category are suppressed (that is, no performance-related warning messages are displayed).

To adequately test the compile procedure behavior, the MY_PACKAGE package should contain a subprogram with a collection (PL/SQL table) specified as an OUT or IN OUT argument without using the NOCOPY hint. Normally, with the PERFORMANCE category enabled, a compiler warning will be issued. Using the code examples shown in the last two slides, the warnings related to the NOCOPY hint are suppressed.

Summary

In this lesson, you should have learned how to:

- **Switch between native and interpreted compilations**
- **Set parameters that influence native compilation of PL/SQL programs**
- **Query data dictionary views that provide information on PL/SQL compilation settings**
- **Use the PL/SQL compiler warning mechanism:**
  - **Declaratively by setting the `PLSQL_WARNINGS` parameter**
  - **Programmatically using the `DBMS_WARNING` package**

Summary

The lesson covers details about how native and interpreted compilations work and how to use parameters that influence the way PL/SQL code is compiled.

The key recommendation is to enable native compilation by default, resulting in 30% faster performance (in some cases) for your PL/SQL logic. Benchmarks have shown that enabling native compilation in Oracle Database 10*g* results in twice the performance when compared to Oracle8*i* and Oracle9*i* databases, and as much as three times the performance of PL/SQL code executing in an Oracle8 database environment. For more information, refer to the Oracle white paper titled "PL/SQL Just Got Faster," by Bryn Llewellyn and Charles Wetherell, from the Oracle Technology Network (OTN) Web site at http://otn.oracle.com.

The lesson also covers the following two ways of influencing the new compiler warning system that was added to Oracle Database 10*g*:

- Setting the `PLSQL_WARNINGS` parameter
- Using the `DBMS_WARNING` package programmatic interface

Practice 12: Overview

This practice covers the following topics:

- Enabling native compilation for your session and compiling a procedure
- Creating a subprogram to compile a PL/SQL procedure, function, or a package; suppressing warnings for the `PERFORMANCE` compiler warning category; and restoring the original session warning settings
- Executing the procedure to compile a PL/SQL package containing a procedure that uses a PL/SQL table as an `IN OUT` parameter without specifying the `NOCOPY` hint

ORACLE

Practice 12: Overview

In this practice, you enable native compilation for your session and compile a procedure. You then create a subprogram to compile a PL/SQL procedure, function, or a package, and you suppress warnings for the `PERFORMANCE` compiler warning category. The procedure must restore the original session warning settings. You then execute the procedure to compile a PL/SQL package that you create, where the package contains a procedure with an `IN OUT` parameter without specifying the `NOCOPY` hint.

Practice 12

1. Alter the PLSQL_COMPILER_FLAGS parameter to enable native compilation for your session, and compile any subprogram that you have written.
    a. Execute the ALTER SESSION command to enable native compilation.
    b. Compile the EMPLOYEE_REPORT procedure. What occurs during compilation?
    c. Execute the EMPLOYEE_REPORT with the value 'UTL_FILE' as the first parameter, and 'native_salrepXX.txt' where XX is your student number.
    d. Switch compilation to use interpreted compilation.
2. In the COMPILE_PKG (from Practice 6), add an overloaded version of the procedure called MAKE, which will compile a named procedure, function, or package.
    a. In the specification, declare a MAKE procedure that accepts two string arguments, one for the name of the PL/SQL construct and the other for the type of PL/SQL program, such as PROCEDURE, FUNCTION, PACKAGE, or PACKAGE BODY.
    b. In the body, write the MAKE procedure to call the DBMS_WARNINGS package to suppress the PERFORMANCE category. However, save the current compiler warning settings before you alter them. Then write an EXECUTE IMMEDIATE statement to compile the PL/SQL object using an appropriate ALTER...COMPILE statement with the supplied parameter values. Finally, restore the compiler warning settings that were in place for the calling environment before the procedure is invoked.
3. Write a new PL/SQL package called TEST_PKG containing a procedure called GET_EMPLOYEES that uses an IN OUT argument.
    a. In the specification, declare the GET_EMPLOYEES procedure with two parameters: an input parameter specifying a department ID, and an IN OUT parameter specifying a PL/SQL table of employee rows.
    Hint: You must declare a TYPE in the package specification for the PL/SQL table parameter's data type.
    b. In the package body, implement the GET_EMPLOYEES procedure to retrieve all the employee rows for a specified department into the PL/SQL table IN OUT parameter.
    Hint: Use the SELECT … BULK COLLECT INTO syntax to simplify the code.
4. Use the ALTER SESSION statement to set the PLSQL_WARNINGS so that all compiler warning categories are enabled.
5. Recompile the TEST_PKG that you created two steps earlier (in Exercise 3). What compiler warnings are displayed, if any?
6. Write a PL/SQL anonymous block to compile the TEST_PKG package by using the overloaded COMPILE_PKG.MAKE procedure with two parameters. The anonymous block should display the current session warning string value before and after it invokes the COMPILE_PKG.MAKE procedure. Do you see any warning messages? Confirm your observations by executing the SHOW ERRORS PACKAGE command for the TEST_PKG.

A

Practice Solutions

Practice I: Solutions

1. Launch *i*SQL*Plus using the icon provided on your desktop.

   a. Log in to the database by using the username and database connect string details provided by your instructor (optionally, write the information here for your records):
   Username: **`ora__`**
   Password: **`ora__`**
   Database Connect String/Tnsname: **t1**

   b. Execute basic `SELECT` statements to query the data in the `DEPARTMENTS`, `EMPLOYEES`, and `JOBS` tables. Take a few minutes to familiarize yourself with the data, or consult Appendix B, which provides a description and some data from each table in the Human Resources schema.

```
SELECT * FROM departments;
SELECT * FROM employees;
```

2. Create a procedure called `HELLO` to display the text `Hello World`.

   a. Create a procedure called `HELLO`.

   b. In the executable section, use the `DBMS_OUTPUT.PUT_LINE` procedure to print `Hello World`, and save the code in the database.
   Note: If you get compile-time errors, then edit the PL/SQL to correct the code, and replace the `CREATE` keyword with the text `CREATE OR REPLACE`.

```
CREATE PROCEDURE hello IS
BEGIN
  DBMS_OUTPUT.PUT_LINE('Hello World');
END;
/

Procedure created.
```

   c. Execute the `SET SERVEROUTPUT ON` command to ensure that the output from the `DBMS_OUTPUT.PUT_LINE` procedure will be displayed in *i*SQL*Plus.

```
SET SERVEROUTPUT ON
```

   d. Create an anonymous block to invoke the stored procedure.

```
BEGIN
  hello;
END;
/

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

Practice I: Solutions (continued)

3. Create a function called TOTAL_SALARY to compute the sum of all employee salaries.

   a. Create a function called TOTAL_SALARY that returns a NUMBER.

   b. In the executable section, execute a query to store the total salary of all employees in a local variable that you declare in the declaration section. Return the value stored in the local variable. Compile the code.

```
CREATE FUNCTION total_salary RETURN NUMBER IS
  total employees.salary%type;
BEGIN
  SELECT SUM(salary) INTO total
  FROM employees;
  RETURN total;
END;
/

Function created.
```

   c. Use an anonymous block to invoke the function. To display the result computed by the function, use the DBMS_OUTPUT.PUT_LINE procedure.
   Hint: Either nest the function call inside the DBMS_OUTPUT.PUT_LINE parameter, or store the function result in a local variable of the anonymous block and use the local variable in the DBMS_OUTPUT.PUT_LINE procedure.

```
DECLARE
  total number := total_salary;
BEGIN
  DBMS_OUTPUT.PUT_LINE('Total Salary: '|| total);
END;
/
-- OR ...
BEGIN
  DBMS_OUTPUT.PUT_LINE('Total Salary: '|| total_salary);
END;
/

Total Salary: 691400
PL/SQL procedure successfully completed.

Total Salary: 691400
PL/SQL procedure successfully completed.
```

Practice I: Solutions (continued)

If you have time, complete the following exercise:

4. Launch SQL*Plus using the icon that is provided on your desktop.

   a. Invoke the procedure and function that you created in exercises 2 and 3.

```
SET SERVEROUTPUT ON
EXECUTE hello;

Hello World
PL/SQL procedure successfully completed.

EXECUTE DBMS_OUTPUT.PUT_LINE('Total Salary: '|| total_salary);

Total Salary: 691400
PL/SQL procedure successfully completed.
```

   b. Create a new procedure called `HELLO_AGAIN` to print `Hello World again`.

```
CREATE PROCEDURE hello_again IS
BEGIN
  DBMS_OUTPUT.PUT_LINE('Hello World again');
END;
/

Procedure created.
```

   c. Invoke the `HELLO_AGAIN` procedure with an anonymous block.

```
SET SERVEROUTPUT ON
BEGIN
  hello_again;
END;
/

Hello World again
PL/SQL procedure successfully completed.
```

Practice 1: Solutions

Note: You can find table descriptions and sample data in Appendix B, "Table Descriptions and Data." Click the Save Script button to save your subprograms as `.sql` files in your local file system.
Remember to enable `SERVEROUTPUT` if you have previously disabled it.

1. Create and invoke the `ADD_JOB` procedure and consider the results.

   a. Create a procedure called `ADD_JOB` to insert a new job into the `JOBS` table. Provide the ID and title of the job using two parameters.

```
CREATE OR REPLACE PROCEDURE add_job (
  jobid jobs.job_id%TYPE,
  jobtitle jobs.job_title%TYPE) IS
BEGIN
  INSERT INTO jobs (job_id, job_title)
  VALUES (jobid, jobtitle);
  COMMIT;
END add_job;
/

Procedure created.
```

   b. Compile the code, and invoke the procedure with `IT_DBA` as job ID and `Database Administrator` as job title. Query the `JOBS` table to view the results.

```
EXECUTE add_job ('IT_DBA', 'Database Administrator')
SELECT * FROM jobs WHERE job_id = 'IT_DBA';

PL/SQL Procedure Successfully Completed.
```

JOB_ID	JOB_TITLE	MIN_SALARY	MAX_SALARY
IT_DBA	Database Administrator		

   c. Invoke your procedure again, passing a job ID of `ST_MAN` and a job title of `Stock Manager`. What happens and why?

```
EXECUTE add_job ('ST_MAN', 'Stock Manager')

BEGIN add_job ('ST_MAN', 'Stock Manager'); END;

*

ERROR at line 1:
ORA-00001: unique constraint (ORA1.JOB_ID_PK) violated
ORA-06512: at "ORA1.ADD_JOB", line 5
ORA-06512: at line 1
```

An exception occurs because there is a primary key integrity constraint on the `JOB_ID` column.

Practice 1: Solutions (continued)

2. Create a procedure called UPD_JOB to modify a job in the JOBS table.

   a. Create a procedure called UPD_JOB to update the job title. Provide the job ID and a new title using two parameters. Include the necessary exception handling if no update occurs.

```
CREATE OR REPLACE PROCEDURE upd_job(
  jobid IN jobs.job_id%TYPE,
  jobtitle IN jobs.job_title%TYPE) IS
BEGIN
  UPDATE jobs
  SET    job_title = jobtitle
  WHERE  job_id = jobid;
  IF SQL%NOTFOUND THEN
    RAISE_APPLICATION_ERROR(-20202, 'No job updated.');
  END IF;
END upd_job;
/

Procedure created.
```

   b. Compile the code; invoke the procedure to change the job title of the job ID IT_DBA to Data Administrator. Query the JOBS table to view the results.

```
EXECUTE upd_job ('IT_DBA', 'Data Administrator')
SELECT * FROM jobs WHERE job_id = 'IT_DBA';

PL/SQL Procedure Successfully Completed.
```

JOB_ID	JOB_TITLE	MIN_SALARY	MAX_SALARY
IT_DBA	Data Administrator		

   Also check the exception handling by trying to update a job that does not exist. (You can use the job ID IT_WEB and the job title Web Master.)

```
EXECUTE upd_job ('IT_WEB', 'Web Master')

BEGIN upd_job ('IT_WEB', 'Web Master'); END;

*

ERROR at line 1:
ORA-20202: No job updated.
ORA-06512: at "ORA1.UPD_JOB", line 9
ORA-06512: at line 1
```

3. Create a procedure called DEL_JOB to delete a job from the JOBS table.

   a. Create a procedure called DEL_JOB to delete a job. Include the necessary exception handling if no job is deleted.

```
CREATE OR REPLACE PROCEDURE del_job (jobid jobs.job_id%TYPE) IS
BEGIN
  DELETE FROM jobs
  WHERE  job_id = jobid;
  IF SQL%NOTFOUND THEN
    RAISE_APPLICATION_ERROR(-20203, 'No jobs deleted.');
  END IF;
END DEL_JOB;
/

Procedure created.
```

   b. Compile the code; invoke the procedure using job ID IT_DBA. Query the JOBS table to view the results.

```
EXECUTE del_job ('IT_DBA')
SELECT * FROM jobs WHERE job_id = 'IT_DBA';

PL/SQL procedure successfully completed.

no rows selected
```

   Also, check the exception handling by trying to delete a job that does not exist. (Use the IT_WEB job ID.) You should get the message that you used in the exception-handling section of the procedure as output.

```
EXECUTE del_job ('IT_WEB')

BEGIN del_job ('IT_WEB'); END;

*

ERROR at line 1:
ORA-20203: No jobs deleted.
ORA-06512: at "ORA1.DEL_JOB", line 6
ORA-06512: at line 1
```

4. Create a procedure called GET_EMPLOYEE to query the EMPLOYEES table, retrieving the salary and job ID for an employee when provided with the employee ID.

   a. Create a procedure that returns a value from the SALARY and JOB_ID columns for a specified employee ID. Compile the code and remove the syntax errors.

Practice 1: Solutions (continued)

```
CREATE OR REPLACE PROCEDURE get_employee
    (empid IN  employees.employee_id%TYPE,
     sal   OUT employees.salary%TYPE,
     job   OUT employees.job_id%TYPE) IS
BEGIN
  SELECT  salary, job_id
  INTO    sal, job
  FROM    employees
  WHERE   employee_id = empid;
END get_employee;
/

Procedure created.
```

b. Execute the procedure using host variables for the two `OUT` parameters, one for the salary and the other for the job ID. Display the salary and job ID for employee ID 120.

```
VARIABLE salary NUMBER
VARIABLE job    VARCHAR2(15)
EXECUTE get_employee(120, :salary, :job)
PRINT salary job

PL/SQL procedure successfully completed.
```

SALARY
8000

JOB
ST_MAN

c. Invoke the procedure again, passing an `EMPLOYEE_ID` of 300. What happens and why?

```
EXECUTE get_employee(300, :salary, :job)

BEGIN get_employee(300, :salary, :job); END;

*

ERROR at line 1:
ORA-01403: no data found
ORA-06512: at "ORA1.GET_EMPLOYEE", line 6
ORA-06512: at line 1
```

There is no employee in the `EMPLOYEES` table with an `EMPLOYEE_ID` of 300. The `SELECT` statement retrieved no data from the database, resulting in a fatal PL/SQL error: `NO_DATA_FOUND`.

Practice 2: Solutions

1. Create and invoke the GET_JOB function to return a job title.

   a. Create and compile a function called GET_JOB to return a job title.

```
CREATE OR REPLACE FUNCTION get_job (jobid IN jobs.job_id%type )
RETURN jobs.job_title%type IS
  title jobs.job_title%type;
BEGIN
  SELECT job_title
  INTO title
  FROM jobs
  WHERE job_id = jobid;
  RETURN title;
END get_job;
/

Function created.
```

   b. Create a VARCHAR2 host variable called TITLE, allowing a length of 35 characters. Invoke the function with SA_REP job ID to return the value in the host variable. Print the host variable to view the result.

```
VARIABLE title VARCHAR2(35)
EXECUTE :title := get_job ('SA_REP');
PRINT title

PL/SQL procedure successfully completed.
```

TITLE
Sales Representative

2. Create a function called GET_ANNUAL_COMP to return the annual salary computed from an employee's monthly salary and commission passed as parameters.

   a. Develop and store the function GET_ANNUAL_COMP, accepting parameter values for monthly salary and commission. Either or both values passed can be NULL, but the function should still return a non-NULL annual salary. Use the following basic formula to calculate the annual salary:

      (salary*12) + (commission_pct*salary*12)

```
CREATE OR REPLACE FUNCTION get_annual_comp(
  sal  IN employees.salary%TYPE,
  comm IN employees.commission_pct%TYPE)
 RETURN NUMBER IS
BEGIN
  RETURN (NVL(sal,0) * 12 + (NVL(comm,0) * nvl(sal,0) * 12));
END get_annual_comp;
/

Function created.
```

Practice 2: Solutions (continued)

b. Use the function in a SELECT statement against the EMPLOYEES table for employees in department 30.

```
SELECT employee_id, last_name,
       get_annual_comp(salary,commission_pct) "Annual Compensation"
FROM   employees
WHERE department_id=30
/
```

EMPLOYEE_ID	LAST_NAME	Annual Compensation
114	Raphaely	132000
115	Khoo	37200
116	Baida	34800
117	Tobias	33600
118	Himuro	31200
119	Colmenares	30000

6 rows selected.

3. Create a procedure, ADD_EMPLOYEE, to insert a new employee into the EMPLOYEES table. The procedure should call a VALID_DEPTID function to check whether the department ID specified for the new employee exists in the DEPARTMENTS table.

   a. Create a function VALID_DEPTID to validate a specified department ID and return a BOOLEAN value of TRUE if the department exists.

```
CREATE OR REPLACE FUNCTION valid_deptid(
  deptid IN departments.department_id%TYPE)
  RETURN BOOLEAN IS
  dummy  PLS_INTEGER;
BEGIN
  SELECT  1
  INTO    dummy
  FROM    departments
  WHERE   department_id = deptid;
  RETURN  TRUE;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    RETURN FALSE;
END valid_deptid;
/

Function created.
```

Practice 2: Solutions (continued)

b. Create the ADD_EMPLOYEE procedure to add an employee to the EMPLOYEES table. The row should be added to the EMPLOYEES table if the VALID_DEPTID function returns TRUE; otherwise, alert the user with an appropriate message. Provide the following parameters (with defaults specified in parentheses): first_name, last_name, email, job (SA_REP), mgr (145), sal (1000), comm (0), and deptid (30). Use the EMPLOYEES_SEQ sequence to set the employee_id column, and set hire_date to TRUNC(SYSDATE).

```
CREATE OR REPLACE PROCEDURE add_employee(
   first_name employees.first_name%TYPE,
   last_name  employees.last_name%TYPE,
   email      employees.email%TYPE,
   job        employees.job_id%TYPE         DEFAULT 'SA_REP',
   mgr        employees.manager_id%TYPE     DEFAULT 145,
   sal        employees.salary%TYPE         DEFAULT 1000,
   comm       employees.commission_pct%TYPE DEFAULT 0,
   deptid     employees.department_id%TYPE  DEFAULT 30) IS
BEGIN
 IF valid_deptid(deptid) THEN
   INSERT INTO employees(employee_id, first_name, last_name, email,
     job_id, manager_id, hire_date, salary, commission_pct,
department_id)
   VALUES (employees_seq.NEXTVAL, first_name, last_name, email,
     job, mgr, TRUNC(SYSDATE), sal, comm, deptid);
 ELSE
   RAISE_APPLICATION_ERROR (-20204, 'Invalid department ID. Try again.');
 END IF;
END add_employee;
/

Procedure created.
```

c. Call ADD_EMPLOYEE for the name Jane Harris in department 15, leaving other parameters with their default values. What is the result?

Note: If the database server time is not between 8:00 and 18:00, the Secure_employees trigger will be fired on performing any DML operation on the EMPLOYEES table. Disable the aforesaid trigger to overcome this problem.

```
EXECUTE add_employee('Jane', 'Harris', 'JAHARRIS', deptid=> 15)

BEGIN add_employee('Jane', 'Harris', 'JAHARRIS', deptid=> 15); END;

*

ERROR at line 1:
ORA-20204: Invalid department ID. Try again.
ORA-06512: at "ORA1.ADD_EMPLOYEE", line 17
ORA-06512: at line 1
```

d. Add another employee named `Joe Harris` in department 80, leaving the remaining parameters with their default values. What is the result?

```
EXECUTE add_employee('Joe', 'Harris', 'JAHARRIS', deptid=> 80)

PL/SQL procedure successfully completed.
```

Practice 3: Solutions

1. Create a package specification and body called `JOB_PKG`, containing a copy of your `ADD_JOB`, `UPD_JOB`, and `DEL_JOB` procedures, as well as your `GET_JOB` function.
   Tip: Consider saving the package specification and body in two separate files (for example, `p3q1_s.sql` and `p3q1_b.sql` for the package specification and body, respectively). Include a `SHOW ERRORS` statement after the `CREATE PACKAGE` statement in each file. Alternatively, place all code in one file.
   Note: Use the code in your previously saved script files when creating the package.

   a. Create the package specification including the procedures and function headings as public constructs.

```
CREATE OR REPLACE PACKAGE job_pkg IS
  PROCEDURE add_job (jobid jobs.job_id%TYPE, jobtitle
jobs.job_title%TYPE);
  PROCEDURE del_job (jobid jobs.job_id%TYPE);
  FUNCTION get_job (jobid IN jobs.job_id%type) RETURN
jobs.job_title%type;
  PROCEDURE upd_job(jobid IN jobs.job_id%TYPE, jobtitle IN
jobs.job_title%TYPE);
END job_pkg;
/
SHOW ERRORS

Package created.

No errors.
```

   Note: Consider whether you still need the stand-alone procedures and functions you just packaged.

   b. Create the package body with the implementations for each of the subprograms.

```
CREATE OR REPLACE PACKAGE BODY job_pkg IS
  PROCEDURE add_job (
    jobid jobs.job_id%TYPE,
   jobtitle jobs.job_title%TYPE) IS
  BEGIN
    INSERT INTO jobs (job_id, job_title)
    VALUES (jobid, jobtitle);
    COMMIT;
  END add_job;

  PROCEDURE del_job (jobid jobs.job_id%TYPE) IS
  BEGIN
    DELETE FROM jobs
    WHERE job_id = jobid;
    IF SQL%NOTFOUND THEN
      RAISE_APPLICATION_ERROR(-20203, 'No jobs deleted.');
    END IF;
  END DEL_JOB;
```

```
  FUNCTION get_job (jobid IN jobs.job_id%type)
    RETURN jobs.job_title%type IS
    title jobs.job_title%type;
  BEGIN
    SELECT job_title
    INTO title
    FROM jobs
    WHERE job_id = jobid;
    RETURN title;
  END get_job;

  PROCEDURE upd_job(
    jobid IN jobs.job_id%TYPE,
    jobtitle IN jobs.job_title%TYPE) IS
  BEGIN
    UPDATE jobs
    SET job_title = jobtitle
    WHERE job_id = jobid;
    IF SQL%NOTFOUND THEN
      RAISE_APPLICATION_ERROR(-20202, 'No job updated.');
    END IF;
  END upd_job;

END job_pkg;
/
SHOW ERRORS

Package body created.

No errors.
```

c. Invoke your `ADD_JOB` package procedure by passing the values `IT_SYSAN` and `Systems Analyst` as parameters.

```
EXECUTE job_pkg.add_job('IT_SYSAN', 'Systems Analyst')

PL/SQL procedure successfully completed.
```

d. Query the `JOBS` table to see the result.

```
SELECT *
FROM jobs
WHERE job_id = 'IT_SYSAN';
```

JOB_ID	JOB_TITLE	MIN_SALARY	MAX_SALARY
IT_SYSAN	Systems Analyst		

Practice 3: Solutions (continued)

2. Create and invoke a package that contains private and public constructs.

   a. Create a package specification and package body called `EMP_PKG` that contains your `ADD_EMPLOYEE` and `GET_EMPLOYEE` procedures as public constructs, and include your `VALID_DEPTID` function as a private construct.

   Package specification:

```
CREATE OR REPLACE PACKAGE emp_pkg IS
  PROCEDURE add_employee(
    first_name employees.first_name%TYPE,
    last_name employees.last_name%TYPE,
    email employees.email%TYPE,
    job employees.job_id%TYPE DEFAULT 'SA_REP',
    mgr employees.manager_id%TYPE DEFAULT 145,
    sal employees.salary%TYPE DEFAULT 1000,
    comm employees.commission_pct%TYPE DEFAULT 0,
    deptid employees.department_id%TYPE DEFAULT 30);
PROCEDURE get_employee(
    empid IN employees.employee_id%TYPE,
    sal OUT employees.salary%TYPE,
    job OUT employees.job_id%TYPE);
END emp_pkg;
/
SHOW ERRORS

Package created.

No errors.
```

   Package body:

```
CREATE OR REPLACE PACKAGE BODY emp_pkg IS
  FUNCTION valid_deptid(deptid IN departments.department_id%TYPE)
    RETURN BOOLEAN IS
    dummy PLS_INTEGER;
  BEGIN
    SELECT 1
    INTO dummy
    FROM departments
    WHERE department_id = deptid;
    RETURN TRUE;
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
    RETURN FALSE;
  END valid_deptid;
  -- ...
```

Practice 3: Solutions (continued)

Package body (continued):

```
  PROCEDURE add_employee(
    first_name employees.first_name%TYPE,
    last_name employees.last_name%TYPE,
    email employees.email%TYPE,
    job employees.job_id%TYPE DEFAULT 'SA_REP',
    mgr employees.manager_id%TYPE DEFAULT 145,
    sal employees.salary%TYPE DEFAULT 1000,
    comm employees.commission_pct%TYPE DEFAULT 0,
    deptid employees.department_id%TYPE DEFAULT 30) IS
  BEGIN
    IF valid_deptid(deptid) THEN
      INSERT INTO employees(employee_id, first_name, last_name, email,
       job_id,manager_id,hire_date,salary,commission_pct,department_id)
      VALUES (employees_seq.NEXTVAL, first_name, last_name, email,
        job, mgr, TRUNC(SYSDATE), sal, comm, deptid);
    ELSE
      RAISE_APPLICATION_ERROR (-20204,
                               'Invalid department ID. Try again.');
    END IF;
  END add_employee;

  PROCEDURE get_employee(
    empid IN employees.employee_id%TYPE,
    sal OUT employees.salary%TYPE,
    job OUT employees.job_id%TYPE) IS
  BEGIN
    SELECT salary, job_id
    INTO sal, job
    FROM employees
    WHERE employee_id = empid;
  END get_employee;
END emp_pkg;
/
SHOW ERRORS

Package body created.

No errors.
```

Practice 3: Solutions (continued)

b. Invoke the `EMP_PKG.GET_EMPLOYEE` procedure, using department ID 15 for employee `Jane Harris` with e-mail `JAHARRIS`. Because department ID 15 does not exist, you should get an error message as specified in the exception handler of your procedure.

```
EXECUTE emp_pkg.add_employee('Jane', 'Harris','JAHARRIS', deptid => 15)

BEGIN emp_pkg.add_employee('Jane', 'Harris','JAHARRIS', deptid => 15);
END;

*

ERROR at line 1:
ORA-20204: Invalid department ID. Try again.
ORA-06512: at "ORA1.EMP_PKG", line 31
ORA-06512: at line 1
```

c. Invoke the `GET_EMPLOYEE` package procedure by using department ID 80 for employee `David Smith` with e-mail `DASMITH`.

```
EXECUTE emp_pkg.add_employee('David', 'Smith','DASMITH', deptid => 80)

PL/SQL procedure successfully completed.
```

Note: If you are using SQL Developer, your compile time errors are displayed in the Message Log. If you are using SQL*Plus or iSQL*Plus to create your stored code, use the SQL*Plus `SHOW ERRORS` to view compile errors.

Practice 4: Solutions

1. Copy and modify the code for the EMP_PKG package that you created in Practice 3, Exercise 2, and overload the ADD_EMPLOYEE procedure.

    a. In the package specification, add a new procedure called ADD_EMPLOYEE, which accepts three parameters: the first name, last name, and department ID. Compile the changes.

```
CREATE OR REPLACE PACKAGE emp_pkg IS
  PROCEDURE add_employee(
    first_name employees.first_name%TYPE,
    last_name employees.last_name%TYPE,
    email employees.email%TYPE,
    job employees.job_id%TYPE DEFAULT 'SA_REP',
    mgr employees.manager_id%TYPE DEFAULT 145,
    sal employees.salary%TYPE DEFAULT 1000,
    comm employees.commission_pct%TYPE DEFAULT 0,
    deptid employees.department_id%TYPE DEFAULT 30);
  PROCEDURE add_employee(
    first_name employees.first_name%TYPE,
    last_name employees.last_name%TYPE,
    deptid employees.department_id%TYPE);
  PROCEDURE get_employee(
    empid IN employees.employee_id%TYPE,
    sal OUT employees.salary%TYPE,
    job OUT employees.job_id%TYPE);
END emp_pkg;
/
SHOW ERRORS

Package created.

No errors.
```

    b. Implement the new ADD_EMPLOYEE procedure in the package body so that it formats the e-mail address in uppercase characters, using the first letter of the first name concatenated with the first seven letters of the last name. The procedure should call the existing ADD_EMPLOYEE procedure to perform the actual INSERT operation using its parameters and formatted e-mail to supply the values. Compile the changes.

```
CREATE OR REPLACE PACKAGE BODY emp_pkg IS
  FUNCTION valid_deptid(deptid IN departments.department_id%TYPE)
    RETURN BOOLEAN IS
    dummy PLS_INTEGER;
  BEGIN
    SELECT 1
    INTO dummy
    FROM departments
    WHERE department_id = deptid;
    RETURN TRUE;
```

```
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
    RETURN FALSE;
  END valid_deptid;

  PROCEDURE add_employee(
    first_name employees.first_name%TYPE,
    last_name employees.last_name%TYPE,
    email employees.email%TYPE,
    job employees.job_id%TYPE DEFAULT 'SA_REP',
    mgr employees.manager_id%TYPE DEFAULT 145,
    sal employees.salary%TYPE DEFAULT 1000,
    comm employees.commission_pct%TYPE DEFAULT 0,
    deptid employees.department_id%TYPE DEFAULT 30) IS
  BEGIN
    IF valid_deptid(deptid) THEN
      INSERT INTO employees(employee_id, first_name, last_name, email,
       job_id,manager_id,hire_date,salary,commission_pct,department_id)
      VALUES (employees_seq.NEXTVAL, first_name, last_name, email,
        job, mgr, TRUNC(SYSDATE), sal, comm, deptid);
    ELSE
      RAISE_APPLICATION_ERROR (-20204,
                               'Invalid department ID. Try again.');
    END IF;
  END add_employee;

  PROCEDURE add_employee(
    first_name employees.first_name%TYPE,
    last_name employees.last_name%TYPE,
    deptid employees.department_id%TYPE) IS
    email employees.email%type;
  BEGIN
    email := UPPER(SUBSTR(first_name, 1, 1)||SUBSTR(last_name, 1, 7));
    add_employee(first_name, last_name, email, deptid => deptid);
  END;

  PROCEDURE get_employee(
    empid IN employees.employee_id%TYPE,
    sal OUT employees.salary%TYPE,
    job OUT employees.job_id%TYPE) IS
  BEGIN
    SELECT salary, job_id
    INTO sal, job
    FROM employees
    WHERE employee_id = empid;
  END get_employee;
END emp_pkg;
/
SHOW ERRORS
```

c. Invoke the new ADD_EMPLOYEE procedure using the name Samuel Joplin to be added to department 30.

```
EXECUTE emp_pkg.add_employee('Samuel', 'Joplin', 30)

PL/SQL procedure successfully completed.
```

2. In the EMP_PKG package, create two overloaded functions called GET_EMPLOYEE.

a. In the specification, add a GET_EMPLOYEE function that accepts the parameter called emp_id based on the employees.employee_id%TYPE type, and a second GET_EMPLOYEE function that accepts a parameter called family_name of the employees.last_name%TYPE type. Both functions should return an EMPLOYEES%ROWTYPE. Compile the changes.

```
CREATE OR REPLACE PACKAGE emp_pkg IS
  PROCEDURE add_employee(
    first_name employees.first_name%TYPE,
    last_name employees.last_name%TYPE,
    email employees.email%TYPE,
    job employees.job_id%TYPE DEFAULT 'SA_REP',
    mgr employees.manager_id%TYPE DEFAULT 145,
    sal employees.salary%TYPE DEFAULT 1000,
    comm employees.commission_pct%TYPE DEFAULT 0,
    deptid employees.department_id%TYPE DEFAULT 30);
  PROCEDURE add_employee(
    first_name employees.first_name%TYPE,
    last_name employees.last_name%TYPE,
    deptid employees.department_id%TYPE);
  PROCEDURE get_employee(
    empid IN employees.employee_id%TYPE,
    sal OUT employees.salary%TYPE,
    job OUT employees.job_id%TYPE);
  FUNCTION get_employee(emp_id employees.employee_id%type)
    return employees%rowtype;
  FUNCTION get_employee(family_name employees.last_name%type)
    return employees%rowtype;
END emp_pkg;
/
SHOW ERRORS

Package created.

No errors.
```

Practice 4: Solutions (continued)

b. In the package body, implement the first GET_EMPLOYEE function to query an employee by his or her ID, and the second to use the equality operator on the value supplied in the family_name parameter. Compile the changes.

```
CREATE OR REPLACE PACKAGE BODY emp_pkg IS
  FUNCTION valid_deptid(deptid IN departments.department_id%TYPE)
    RETURN BOOLEAN IS
    dummy PLS_INTEGER;
  BEGIN
    SELECT 1
    INTO dummy
    FROM departments
    WHERE department_id = deptid;
    RETURN TRUE;
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
    RETURN FALSE;
  END valid_deptid;

  PROCEDURE add_employee(
    first_name employees.first_name%TYPE,
    last_name employees.last_name%TYPE,
    email employees.email%TYPE,
    job employees.job_id%TYPE DEFAULT 'SA_REP',
    mgr employees.manager_id%TYPE DEFAULT 145,
    sal employees.salary%TYPE DEFAULT 1000,
    comm employees.commission_pct%TYPE DEFAULT 0,
    deptid employees.department_id%TYPE DEFAULT 30) IS
  BEGIN
    IF valid_deptid(deptid) THEN
      INSERT INTO employees(employee_id, first_name, last_name, email,
        job_id, manager_id,hire_date,salary,commission_pct,department_id)
      VALUES (employees_seq.NEXTVAL, first_name, last_name, email,
        job, mgr, TRUNC(SYSDATE), sal, comm, deptid);
    ELSE
      RAISE_APPLICATION_ERROR (-20204,
                              'Invalid department ID. Try again.');
    END IF;
  END add_employee;

  PROCEDURE add_employee(
    first_name employees.first_name%TYPE,
    last_name employees.last_name%TYPE,
    deptid employees.department_id%TYPE) IS
    email employees.email%type;
  BEGIN
    email := UPPER(SUBSTR(first_name, 1, 1)||SUBSTR(last_name, 1, 7));
    add_employee(first_name, last_name, email, deptid => deptid);
  END;
```

Practice 4: Solutions (continued)

```
  PROCEDURE get_employee(
    empid IN employees.employee_id%TYPE,
    sal OUT employees.salary%TYPE,
    job OUT employees.job_id%TYPE) IS
  BEGIN
    SELECT salary, job_id
    INTO sal, job
    FROM employees
    WHERE employee_id = empid;
  END get_employee;

  FUNCTION get_employee(emp_id employees.employee_id%type)
    return employees%rowtype IS
    emprec employees%rowtype;
  BEGIN
    SELECT * INTO emprec
    FROM employees
    WHERE employee_id = emp_id;
    RETURN emprec;
  END;

  FUNCTION get_employee(family_name employees.last_name%type)
    return employees%rowtype IS
    emprec employees%rowtype;
  BEGIN
    SELECT * INTO emprec
    FROM employees
    WHERE last_name = family_name;
    RETURN emprec;
  END;

END emp_pkg;
/
SHOW ERRORS

Package body created.

No errors.
```

Practice 4: Solutions (continued)

c. Add a utility procedure PRINT_EMPLOYEE to the package that accepts an EMPLOYEES%ROWTYPE as a parameter and displays the department_id, employee_id, first_name, last_name, job_id, and salary for an employee on one line, using DBMS_OUTPUT. Compile the changes.

```
CREATE OR REPLACE PACKAGE emp_pkg IS
  PROCEDURE add_employee(
    first_name employees.first_name%TYPE,
    last_name employees.last_name%TYPE,
    email employees.email%TYPE,
    job employees.job_id%TYPE DEFAULT 'SA_REP',
    mgr employees.manager_id%TYPE DEFAULT 145,
    sal employees.salary%TYPE DEFAULT 1000,
    comm employees.commission_pct%TYPE DEFAULT 0,
    deptid employees.department_id%TYPE DEFAULT 30);
  PROCEDURE add_employee(
    first_name employees.first_name%TYPE,
    last_name employees.last_name%TYPE,
    deptid employees.department_id%TYPE);
  PROCEDURE get_employee(
    empid IN employees.employee_id%TYPE,
    sal OUT employees.salary%TYPE,
    job OUT employees.job_id%TYPE);
  FUNCTION get_employee(emp_id employees.employee_id%type)
    return employees%rowtype;
  FUNCTION get_employee(family_name employees.last_name%type)
    return employees%rowtype;
  PROCEDURE print_employee(emprec employees%rowtype);
END emp_pkg;
/
SHOW ERRORS

Prackage created.

No Errors.

CREATE OR REPLACE PACKAGE BODY emp_pkg IS
  FUNCTION valid_deptid(deptid IN departments.department_id%TYPE)
 RETURN BOOLEAN IS
    dummy PLS_INTEGER;
  BEGIN
    SELECT 1
    INTO dummy
    FROM departments
    WHERE department_id = deptid;
    RETURN TRUE;
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
    RETURN FALSE;
END valid_deptid;
```

Practice 4: Solutions (continued)

```
PROCEDURE add_employee(
  first_name employees.first_name%TYPE,
  last_name employees.last_name%TYPE,
  email employees.email%TYPE,
  job employees.job_id%TYPE DEFAULT 'SA_REP',
  mgr employees.manager_id%TYPE DEFAULT 145,
  sal employees.salary%TYPE DEFAULT 1000,
  comm employees.commission_pct%TYPE DEFAULT 0,
  deptid employees.department_id%TYPE DEFAULT 30) IS
BEGIN
  IF valid_deptid(deptid) THEN
    INSERT INTO employees(employee_id, first_name, last_name, email,
      job_id,manager_id,hire_date,salary,commission_pct,department_id)
    VALUES (employees_seq.NEXTVAL, first_name, last_name, email,
      job, mgr, TRUNC(SYSDATE), sal, comm, deptid);
  ELSE
    RAISE_APPLICATION_ERROR (-20204,
                             'Invalid department ID. Try again.');
  END IF;
END add_employee;

PROCEDURE add_employee(
  first_name employees.first_name%TYPE,
  last_name employees.last_name%TYPE,
  deptid employees.department_id%TYPE) IS
  email employees.email%type;
BEGIN
  email := UPPER(SUBSTR(first_name, 1, 1)||SUBSTR(last_name, 1, 7));
  add_employee(first_name, last_name, email, deptid => deptid);
END;

PROCEDURE get_employee(
  empid IN employees.employee_id%TYPE,
  sal OUT employees.salary%TYPE,
  job OUT employees.job_id%TYPE) IS
BEGIN
  SELECT salary, job_id
  INTO sal, job
  FROM employees
  WHERE employee_id = empid;
END get_employee;

FUNCTION get_employee(emp_id employees.employee_id%type)
  return employees%rowtype IS
  emprec employees%rowtype;
BEGIN
  SELECT * INTO emprec
  FROM employees
  WHERE employee_id = emp_id;
  RETURN emprec;
END;
```

Practice 4: Solutions (continued)

```
  FUNCTION get_employee(family_name employees.last_name%type)
    return employees%rowtype IS
    emprec employees%rowtype;
  BEGIN
    SELECT * INTO emprec
    FROM employees
    WHERE last_name = family_name;
    RETURN emprec;
  END;

  PROCEDURE print_employee(emprec employees%rowtype) IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE(emprec.department_id ||' '||
                         emprec.employee_id||' '||
                         emprec.first_name||' '||
                         emprec.last_name||' '||
                         emprec.job_id||' '||
                         emprec.salary);
  END;
END emp_pkg;
/
SHOW ERRORS

Package body created.

No errors.
```

d. Use an anonymous block to invoke the `EMP_PKG.GET_EMPLOYEE` function with an employee ID of `100`, and family name of `'Joplin'`. Use the `PRINT_EMPLOYEE` procedure to display the results for each row returned.

```
BEGIN
  emp_pkg.print_employee(emp_pkg.get_employee(100));
  emp_pkg.print_employee(emp_pkg.get_employee('Joplin'));
END;
/

90 100 Steven King AD_PRES 24000
30 209 Samuel Joplin SA_REP 1000

PL/SQL procedure successfully completed.
```

Note: The employee ID `209` for Samuel Joplin is allocated by using an Oracle sequence object. You may receive a different value when you execute the PL/SQL block shown in this solution.

Practice 4: Solutions (continued)

3. Because the company does not frequently change its departmental data, you improve performance of your EMP_PKG by adding a public procedure INIT_DEPARTMENTS to populate a private PL/SQL table of valid department IDs. Modify the VALID_DEPTID function to use the private PL/SQL table contents to validate department ID values.

   a. In the package specification, create a procedure called INIT_DEPARTMENTS with no parameters.

```
CREATE OR REPLACE PACKAGE emp_pkg IS
  PROCEDURE add_employee(
    first_name employees.first_name%TYPE,
    last_name employees.last_name%TYPE,
    email employees.email%TYPE,
    job employees.job_id%TYPE DEFAULT 'SA_REP',
    mgr employees.manager_id%TYPE DEFAULT 145,
    sal employees.salary%TYPE DEFAULT 1000,
    comm employees.commission_pct%TYPE DEFAULT 0,
    deptid employees.department_id%TYPE DEFAULT 30);
  PROCEDURE add_employee(
    first_name employees.first_name%TYPE,
    last_name employees.last_name%TYPE,
    deptid employees.department_id%TYPE);
  PROCEDURE get_employee(
    empid IN employees.employee_id%TYPE,
    sal OUT employees.salary%TYPE,
    job OUT employees.job_id%TYPE);
  FUNCTION get_employee(emp_id employees.employee_id%type)
    return employees%rowtype;
  FUNCTION get_employee(family_name employees.last_name%type)
    return employees%rowtype;
  PROCEDURE init_departments;
  PROCEDURE print_employee(emprec employees%rowtype);
END emp_pkg;
/
SHOW ERRORS

Package created.

No errors.
```

   b. In the package body, implement the INIT_DEPARTMENTS procedure to store all department IDs in a private PL/SQL index-by table named valid_departments containing BOOLEAN values. Use the department_id column value as the index to create the entry in the index-by table to indicate its presence, and assign the entry a value of TRUE. Declare the valid_departments variable and its type definition boolean_tabtype before all procedures in the body.

Practice 4: Solutions (continued)

```
CREATE OR REPLACE PACKAGE BODY emp_pkg IS
  TYPE boolean_tabtype IS TABLE OF BOOLEAN
    INDEX BY BINARY_INTEGER;
  valid_departments boolean_tabtype;

  FUNCTION valid_deptid(deptid IN departments.department_id%TYPE)
  RETURN BOOLEAN IS
    dummy PLS_INTEGER;
  BEGIN
    ...
  END valid_deptid;

  PROCEDURE add_employee(
    first_name employees.first_name%TYPE,
    last_name employees.last_name%TYPE,
    email employees.email%TYPE,
    job employees.job_id%TYPE DEFAULT 'SA_REP',
    mgr employees.manager_id%TYPE DEFAULT 145,
    sal employees.salary%TYPE DEFAULT 1000,
    comm employees.commission_pct%TYPE DEFAULT 0,
    deptid employees.department_id%TYPE DEFAULT 30) IS
  BEGIN
    ...
  END add_employee;

  PROCEDURE add_employee(
    first_name employees.first_name%TYPE,
    last_name employees.last_name%TYPE,
    deptid employees.department_id%TYPE) IS
    email employees.email%type;
  BEGIN
    ...
  END;

  PROCEDURE get_employee(
    empid IN employees.employee_id%TYPE,
    sal OUT employees.salary%TYPE,
    job OUT employees.job_id%TYPE) IS
  BEGIN
    ...
  END get_employee;

  FUNCTION get_employee(emp_id employees.employee_id%type)
    return employees%rowtype IS
    emprec employees%rowtype;
  BEGIN
    ...
  END;
```

Practice 4: Solutions (continued)

```
  FUNCTION get_employee(family_name employees.last_name%type)
    return employees%rowtype IS
    emprec employees%rowtype;
  BEGIN
    SELECT * INTO emprec
    FROM employees
    WHERE last_name = family_name;
    RETURN emprec;
  END;

  PROCEDURE print_employee(emprec employees%rowtype) IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE(emprec.department_id ||' '||
                         emprec.employee_id||' '||
                         emprec.first_name||' '||
                         emprec.last_name||' '||
                         emprec.job_id||' '||
                         emprec.salary);
  END;

  PROCEDURE init_departments IS
  BEGIN
    FOR rec IN (SELECT department_id FROM departments)
    LOOP
      valid_departments(rec.department_id) := TRUE;
    END LOOP;
  END;
END emp_pkg;
/
SHOW ERRORS

Package body created.

No errors.
```

c. In the body, create an initialization block that calls the `INIT_DEPARTMENTS` procedure to initialize the table. Compile the changes.

```
CREATE OR REPLACE PACKAGE BODY emp_pkg IS
  ...
  PROCEDURE init_departments IS
  BEGIN
    FOR rec IN (SELECT department_id FROM departments)
    LOOP
      valid_departments(rec.department_id) := TRUE;
    END LOOP;
  END;
BEGIN
  init_departments;
END emp_pkg;
/
SHOW ERRORS
Package body created.

No errors
```

Practice 4: Solutions (continued)

4. Change the `VALID_DEPTID` validation processing to use the private PL/SQL table of department IDs.

   a. Modify `VALID_DEPTID` to perform its validation by using the PL/SQL table of department ID values. Compile the changes.

```
CREATE OR REPLACE PACKAGE BODY emp_pkg IS
  TYPE boolean_tabtype IS TABLE OF BOOLEAN
    INDEX BY BINARY_INTEGER;
  valid_departments boolean_tabtype;

  FUNCTION valid_deptid(deptid IN departments.department_id%TYPE)
   RETURN BOOLEAN IS
    dummy PLS_INTEGER;
  BEGIN
    RETURN valid_departments.exists(deptid);
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
    RETURN FALSE;
  END valid_deptid;
  ...

  PROCEDURE init_departments IS
  BEGIN
    FOR rec IN (SELECT department_id FROM departments)
    LOOP
      valid_departments(rec.department_id) := TRUE;
    END LOOP;
  END;
BEGIN
  init_departments;
END emp_pkg;
/
SHOW ERRORS

Package body created.

No errors.
```

   b. Test your code by calling `ADD_EMPLOYEE` using the name `James Bond` in department 15. What happens?

```
EXECUTE emp_pkg.add_employee('James', 'Bond', 15)

BEGIN emp_pkg.add_employee('James', 'Bond', 15); END;

*
ERROR at line 1:
ORA-20204: Invalid department ID. Try again.
ORA-06512: at "ORA1.EMP_PKG", line 32
ORA-06512: at "ORA1.EMP_PKG", line 43
ORA-06512: at line 1
```

Practice 4: Solutions (continued)

The insert operation to add the employee fails with an exception, because department 15 does not exist.

c. Insert a new department with ID 15 and name Security, and commit the changes.

```
INSERT INTO departments (department_id, department_name)
VALUES (15, 'Security');
COMMIT;

1 row created.

Commit complete.
```

d. Test your code again, by calling `ADD_EMPLOYEE` using the name `James Bond` in department 15. What happens?

```
EXECUTE emp_pkg.add_employee('James', 'Bond', 15)

BEGIN emp_pkg.add_employee('James', 'Bond', 15); END;

*
ERROR at line 1:
ORA-20204: Invalid department ID. Try again.
ORA-06512: at "ORA1.EMP_PKG", line 32
ORA-06512: at "ORA1.EMP_PKG", line 43
ORA-06512: at line 1
```

The insert operation to add the employee fails with an exception because department 15 does not exist as an entry in the PL/SQL index-by table package state variable.

e. Execute the `EMP_PKG.INIT_DEPARTMENTS` procedure to update the internal PL/SQL table with the latest departmental data.

```
EXECUTE EMP_PKG.INIT_DEPARTMENTS

PL/SQL procedure successfully completed.
```

f. Test your code by calling `ADD_EMPLOYEE` using the employee name `James Bond`, who works in department 15. What happens?

```
EXECUTE emp_pkg.add_employee('James', 'Bond', 15)

PL/SQL procedure successfully completed.
```

The row is finally inserted because the department 15 record exists in the database and package PL/SQL index-by table due to invoking `EMP_PKG.INIT_DEPARTMENTS`, which refreshes the package state data.

g. Delete employee James Bond and department 15 from their respective tables, commit the changes, and refresh the department data by invoking the EMP_PKG.INIT_DEPARTMENTS procedure.

```
DELETE FROM employees
WHERE first_name = James AND last_name = Bond;
DELETE FROM departments WHERE department_id = 15;
COMMIT;
EXECUTE EMP_PKG.INIT_DEPARTMENTS

1 row deleted.
1 row deleted.
Commit complete.
PL/SQL procedure successfully completed.
```

5. Reorganize the subprograms in the package specification body so that they are in alphabetical sequence.

   a. Edit the package specification and reorganize subprograms alphabetically. In *i*SQL*Plus, load and compile the package specification. What happens?

```
CREATE OR REPLACE PACKAGE emp_pkg IS
  PROCEDURE add_employee(
    first_name employees.first_name%TYPE,
    last_name employees.last_name%TYPE,
    email employees.email%TYPE,
    job employees.job_id%TYPE DEFAULT 'SA_REP',
    mgr employees.manager_id%TYPE DEFAULT 145,
    sal employees.salary%TYPE DEFAULT 1000,
    comm employees.commission_pct%TYPE DEFAULT 0,
    deptid employees.department_id%TYPE DEFAULT 30);
  PROCEDURE add_employee(
    first_name employees.first_name%TYPE,
    last_name employees.last_name%TYPE,
    deptid employees.department_id%TYPE);
  PROCEDURE get_employee(
    empid IN employees.employee_id%TYPE,
    sal OUT employees.salary%TYPE,
    job OUT employees.job_id%TYPE);
  FUNCTION get_employee(emp_id employees.employee_id%type)
    return employees%rowtype;
  FUNCTION get_employee(family_name employees.last_name%type)
    return employees%rowtype;
  PROCEDURE init_departments;
  PROCEDURE print_employee(emprec employees%rowtype);
END emp_pkg;
/
SHOW ERRORS
```

It compiles successfully.
Note: The package may already have its subprograms in alphabetical sequence.

Practice 4: Solutions (continued)

b. Edit the package body and reorganize all subprograms alphabetically. In *i*SQL*Plus, load and compile the package specification. What happens?

```
CREATE OR REPLACE PACKAGE BODY emp_pkg IS
  TYPE boolean_tabtype IS TABLE OF BOOLEAN
    INDEX BY BINARY_INTEGER;
  valid_departments boolean_tabtype;

  PROCEDURE add_employee(
    first_name employees.first_name%TYPE,
    last_name employees.last_name%TYPE,
    email employees.email%TYPE,
    job employees.job_id%TYPE DEFAULT 'SA_REP',
    mgr employees.manager_id%TYPE DEFAULT 145,
    sal employees.salary%TYPE DEFAULT 1000,
    comm employees.commission_pct%TYPE DEFAULT 0,
    deptid employees.department_id%TYPE DEFAULT 30) IS
  BEGIN
    IF valid_deptid(deptid) THEN
      INSERT INTO employees(employee_id, first_name, last_name, email,
        job_id, manager_id, hire_date, salary, commission_pct,
department_id)
      VALUES (employees_seq.NEXTVAL, first_name, last_name, email,
        job, mgr, TRUNC(SYSDATE), sal, comm, deptid);
    ELSE
      RAISE_APPLICATION_ERROR (-20204, 'Invalid department ID. Try
again.');
    END IF;
  END add_employee;

  PROCEDURE add_employee(
    first_name employees.first_name%TYPE,
    last_name employees.last_name%TYPE,
    deptid employees.department_id%TYPE) IS
    email employees.email%type;
  BEGIN
    email := UPPER(SUBSTR(first_name, 1, 1)||SUBSTR(last_name, 1, 7));
    add_employee(first_name, last_name, email, deptid => deptid);
  END;

  PROCEDURE get_employee(
    empid IN employees.employee_id%TYPE,
    sal OUT employees.salary%TYPE,
    job OUT employees.job_id%TYPE) IS
  BEGIN
    SELECT salary, job_id
    INTO sal, job
    FROM employees
    WHERE employee_id = empid;
  END get_employee;
```

Practice 4: Solutions (continued)

```
FUNCTION get_employee(emp_id employees.employee_id%type)
  return employees%rowtype IS
  emprec employees%rowtype;
BEGIN
  SELECT * INTO emprec
  FROM employees
  WHERE employee_id = emp_id;
  RETURN emprec;
END;

FUNCTION get_employee(family_name employees.last_name%type)
  return employees%rowtype IS
  emprec employees%rowtype;
BEGIN
  SELECT * INTO emprec
  FROM employees
  WHERE last_name = family_name;
  RETURN emprec;
END;

PROCEDURE init_departments IS
BEGIN
  FOR rec IN (SELECT department_id FROM departments)
  LOOP
    valid_departments(rec.department_id) := TRUE;
  END LOOP;
END;

PROCEDURE print_employee(emprec employees%rowtype) IS
BEGIN
  DBMS_OUTPUT.PUT_LINE(emprec.department_id ||' '||
                       emprec.employee_id||' '||
                       emprec.first_name||' '||
                       emprec.last_name||' '||
                       emprec.job_id||' '||
                       emprec.salary);
END;

FUNCTION valid_deptid(deptid IN departments.department_id%TYPE)
 RETURN BOOLEAN IS
  dummy PLS_INTEGER;
BEGIN
  RETURN valid_departments.exists(deptid);
EXCEPTION
  WHEN NO_DATA_FOUND THEN
  RETURN FALSE;
END valid_deptid;
```

```
BEGIN
  init_departments;
END emp_pkg;
/
SHOW ERRORS

Warning: Package Body created with compilation errors.

Errors for PACKAGE BODY EMP_PKG:

LINE/COL ERROR
-------- -----------------------------------------------------------------
16/5     PL/SQL: Statement ignored
16/8     PLS-00313: 'VALID_DEPTID' not declared in this scope
```

It does not compile successfully because the `VALID_DEPTID` function is referenced before it is declared.

c. Fix the compilation error by using a forward declaration in the body for the offending subprogram reference. Load and re-create the package body. What happens? Save the package code in a script file.

```
CREATE OR REPLACE PACKAGE BODY emp_pkg IS
  TYPE boolean_tabtype IS TABLE OF BOOLEAN
    INDEX BY BINARY_INTEGER;
  valid_departments boolean_tabtype;

  FUNCTION valid_deptid(deptid IN departments.department_id%TYPE)
   RETURN BOOLEAN;

  PROCEDURE add_employee(
    first_name employees.first_name%TYPE,
    last_name employees.last_name%TYPE,
    email employees.email%TYPE,
    job employees.job_id%TYPE DEFAULT 'SA_REP',
    mgr employees.manager_id%TYPE DEFAULT 145,
    sal employees.salary%TYPE DEFAULT 1000,
    comm employees.commission_pct%TYPE DEFAULT 0,
    deptid employees.department_id%TYPE DEFAULT 30) IS
  BEGIN
    IF valid_deptid(deptid) THEN
      INSERT INTO employees(employee_id, first_name, last_name, email,
        job_id,manager_id,hire_date,salary,commission_pct,department_id)
      VALUES (employees_seq.NEXTVAL, first_name, last_name, email,
        job, mgr, TRUNC(SYSDATE), sal, comm, deptid);
    ELSE
      RAISE_APPLICATION_ERROR (-20204,
                               'Invalid department ID. Try again.');
    END IF;
  END add_employee;
  ...
```

Practice 4: Solutions (continued)

```
  FUNCTION valid_deptid(deptid IN departments.department_id%TYPE)
   RETURN BOOLEAN IS
    dummy PLS_INTEGER;
  BEGIN
    RETURN valid_departments.exists(deptid);
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
    RETURN FALSE;
  END valid_deptid;

BEGIN
  init_departments;
END emp_pkg;
/
SHOW ERRORS

Package body created.

No errors.
```

A forward declaration for the `VALID_DEPTID` function enables the package body to be compiled successfully.

If you have time, complete the following exercise:

6. Wrap the `EMP_PKG` package body and re-create it.

   a. Query the data dictionary to view the source for the `EMP_PKG` body.

```
SELECT text
FROM user_source
WHERE name = 'EMP_PKG'
AND type = 'PACKAGE BODY'
ORDER BY line;
```

TEXT
PACKAGE BODY emp_pkg IS
TYPE boolean_tabtype IS TABLE OF BOOLEAN
INDEX BY BINARY_INTEGER;
valid_departments boolean_tabtype;
BEGIN
init_departments;
END emp_pkg;

100 rows selected.

b. Start a command window and execute the `WRAP` command-line utility to wrap the body of the `EMP_PKG` package. Give the output file name a `.plb` extension.
Hint: Copy the file (which you saved in step 5c) containing the package body to a file called `emp_pkg_b.sql`.

```
WRAP INAME=emp_pkg_b.sql

PL/SQL Wrapper: Release 10.2.0.1.0- Production on Tue Nov 14 03:49:53
2006

Copyright (c) 1993, 2004,Oracle.  All Rights Reserved.

Processing emp_pkg_b.sql to emp_pkg_b.plb
```

c. Using *i*SQL*Plus, load and execute the `.plb` file containing the wrapped source.

```
CREATE OR REPLACE PACKAGE BODY emp_pkg wrapped
0
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
 :
 :
be 4 0
67 3 0
15 2 0
133 6 0
5 1 0
5d 3 0
193 1 8
0

/
SHOW ERRORS

Package body created.

No errors.
```

Practice 4: Solutions (continued)

d. Query the data dictionary to display the source for the `EMP_PKG` package body again. Are the original source code lines readable?

```
SELECT text
FROM user_source
WHERE name = 'EMP_PKG'
AND type = 'PACKAGE BODY'
ORDER BY line;
```

TEXT
PACKAGE BODY emp_pkg wrapped 0 abcd abcd abcd abcd abcd abcd abcd abcd abcd abcd abcd abcd abcd abcd abcd 3 b 9200000 1 4 0 44 2 :e:
121 2 11b 11c 1 129 1 133 1 13b 1 143 1 14c 1 150 2 14f 150 1 14b 2 158 15b 1 [illegible] 2 180 181 1 17c 2 18[illegible]
[illegible]3e 1 217 1 21a 1 21f 1 21c 1 222 1 2[illegible] 2 a9 f5 12f 18[illegible] 1fc 227 1 4 0 235 0 1 14 b 24 0 1 1 1 1 1 1 8 1 1 0 0 0 0 0 0 0 0 0 ca 4 0 36 3 0 b5 4 0 2d 3 0 1b5 a 0 174 7 0 143 6 0 20e b 0 ab 1 4 23 1 3 10b 5 0 3f 3 0 163 1 7 132 1 6 f7 1 5 49 3 0 197 9 0 101 5 0 53 3 0 1ff 1 b 14 1 2 e 1 0 ac 4 0 24 3 0 f8 5 0 164 7 0 3 0 1 1b4 1 a 200 b 0 be 4 0 67 3 0 15 2 0 133 6 0 5 1 0 5d 3 0 193 1 8 0

The source code for the body is no longer readable. You can view the wrapped source, but the orginal source code is not shown. For this reason, make sure you always have a secure copy of your source code in files outside the database when using the `WRAP` utility.

Practice 5: Solutions

1. Create a procedure called EMPLOYEE_REPORT that generates an employee report in a file in the operating system, using the UTL_FILE package. The report should generate a list of employees who have exceeded the average salary of their department.

   a. Your program should accept two parameters. The first parameter is the output directory. The second parameter is the name of the text file that is written.
   Note: Use the directory location value UTL_FILE. Add an exception-handling section to handle errors that may be encountered when using the UTL_FILE package.

   The following is a sample output from the report file:

```
Employees who earn more than average salary:
REPORT GENERATED ON 26-FEB-04
Hartstein                    20      $13,000.00
Raphaely                     30      $11,000.00
Marvis                       40      $6,500.00
...
*** END OF REPORT ***
```

```
CREATE OR REPLACE PROCEDURE employee_report(
  dir IN VARCHAR2, filename IN VARCHAR2) IS
  f UTL_FILE.FILE_TYPE;
  CURSOR avg_csr IS
    SELECT last_name, department_id, salary
    FROM employees outer
    WHERE salary > (SELECT AVG(salary)
                    FROM  employees inner
                    GROUP BY outer.department_id)
    ORDER BY department_id;
BEGIN
  f := UTL_FILE.FOPEN(dir, filename,'w');
  UTL_FILE.PUT_LINE(f, 'Employees who earn more than average salary: ');
  UTL_FILE.PUT_LINE(f, 'REPORT GENERATED ON ' ||SYSDATE);
  UTL_FILE.NEW_LINE(f);
  FOR emp IN avg_csr
  LOOP
    UTL_FILE.PUT_LINE(f,
    RPAD(emp.last_name, 30) || ' ' ||
    LPAD(NVL(TO_CHAR(emp.department_id,'9999'),'-'), 5) || ' ' ||
    LPAD(TO_CHAR(emp.salary, '$99,999.00'), 12));
  END LOOP;
  UTL_FILE.NEW_LINE(f);
  UTL_FILE.PUT_LINE(f, '*** END OF REPORT ***');
  UTL_FILE.FCLOSE(f);
END employee_report;
/

Procedure created.
```

Practice 5: Solutions (continued)

b. Invoke the program, using the second parameter with a name such as `sal_rptxx.txt`, where `xx` represents your user number (for example, 01, 15, and so on).

```
EXECUTE employee_report('UTL_FILE','sal_rpt01.txt')

PL/SQL Procedure sucessfully completed.
```

Note: The data displays the employee's last name, department ID, and salary. Ask your instructor to provide instructions on how to obtain the report file from the server using the `Putty PSFTP` utility.

After you use `PSTFP` to retrieve your generated file, it should contain something similar to the following example:

```
Employees who earn more than average salary:
REPORT GENERATED ON  16-FEB-04

Hartstein                         20   $13,000.00
Raphaely                          30   $11,000.00
Mavris                            40    $6,500.00
Weiss                             50    $8,000.00
Kaufling                          50    $7,900.00
Fripp                             50    $8,200.00
Vollman                           50    $6,500.00
Hunold                            60    $9,000.00
Baer                              70   $10,000.00
Russell                           80   $14,000.00
Bernstein                         80    $9,500.00
Olsen                             80    $8,000.00
  :
  :
Errazuriz                         80   $12,000.00
Zlotkey                           80   $10,500.00
Cambrault                         80   $11,000.00
King                              90   $24,000.00
Kochhar                           90   $17,000.00
De Haan                           90   $17,000.00
Greenberg                        100   $12,000.00
Faviet                           100    $9,000.00
Chen                             100    $8,200.00
Sciarra                          100    $7,700.00
Urman                            100    $7,800.00
Popp                             100    $6,900.00
Higgins                          110   $12,000.00
Gietz                            110    $8,300.00
Grant                              -    $7,000.00

*** END OF REPORT ***
```

Practice 5: Solutions (continued)

2. Create a new procedure called WEB_EMPLOYEE_REPORT that generates the same data as the EMPLOYEE_REPORT.

   a. First, execute SET SERVEROUTPUT ON, and then execute htp.print('hello') followed by executing OWA_UTIL.SHOWPAGE. The exception messages generated can be ignored.

```
SET SERVEROUTPUT ON
EXECUTE HTP.PRINT('hello')
EXECUTE OWA_UTIL.SHOWPAGE

BEGIN htp.print('hello'); END;

*

ERROR at line 1:
ORA-06502: PL/SQL: numeric or value error
ORA-06512: at "SYS.OWA_UTIL", line 325
ORA-06512: at "SYS.HTP", line 1322
ORA-06512: at "SYS.HTP", line 1397
ORA-06512: at "SYS.HTP", line 1684
ORA-06512: at line 1
PL/SQL procedure successfully completed.
```

   These steps are performed to ensure that the messages are not generated again. However, remember that the HTP package is intended to be used in the Oracle HTTP Server context, not *i*SQL*Plus.

   b. Write the WEB_EMPLOYEE_REPORT procedure using the HTP package to generate an HTML report of employees with a salary greater than the average for their departments. If you know HTML, create an HTML table; otherwise, create simple lines of data.
   Hint: Copy the cursor definition and the FOR loop from the EMPLOYEE_REPORT procedure for the basic structure for your Web report.

```
CREATE OR REPLACE PROCEDURE web_employee_report IS
  CURSOR avg_csr IS
    SELECT last_name, department_id, salary
    FROM employees outer
    WHERE salary > (SELECT AVG(salary)
                    FROM  employees inner
                    GROUP BY outer.department_id)
    ORDER BY department_id;
```

Practice 5: Solutions (continued)

```
BEGIN
  htp.htmlopen;
  htp.headopen;
  htp.title('Employee Salary Report');
  htp.headclose;
  htp.bodyopen;
  htp.header(1, 'Employees who earn more than average salary');
  htp.print('REPORT GENERATED ON' || to_char(SYSDATE, 'DD-MON-YY'));
  htp.br;
  htp.hr;
  htp.tableOpen;
  htp.tablerowOpen;
  htp.tableHeader('Last Name');
  htp.tableHeader('Department');
  htp.tableHeader('Salary');
  htp.tablerowclose;

  FOR emp IN avg_csr
  LOOP
    htp.tablerowOpen;
    htp.tabledata(emp.last_name);
    htp.tabledata(NVL(TO_CHAR(emp.department_id,'9999'),'-'));
    htp.tabledata(TO_CHAR(emp.salary, '$99,999.00'));
    htp.tablerowclose;
  END LOOP;

  htp.tableclose;
  htp.hr;
  htp.print('*** END OF REPORT ***');
  htp.bodyclose;
  htp.htmlclose;
END web_employee_report;
/
show errors


Procedure created.

No errors.
```

c. Execute the procedure using *i*SQL*Plus to generate the HTML data into a server buffer, and execute the `OWA_UTIL.SHOWPAGE` procedure to display contents of the buffer. Remember that `SERVEROUTPUT` should be `ON` before you execute the code.

```
EXECUTE web_employee_report
EXECUTE owa_util.showpage

PL/SQL procedure successfully completed.

:
```

Practice 5: Solutions (continued)

```
<HTML>
<HEAD>
<TITLE>Employee Salary Report</TITLE>
</HEAD>
<BODY>
<H1>Employees who earn more than average salary</H1>
REPORT GENERATED ON16-FEB-04
<BR>
<HR>
<TABLE >
<TR>
<TH>Last Name</TH>
<TH>Department</TH>
<TH>Salary</TH>
</TR>
<TR>
<TD>Hartstein</TD>
<TD> 20</TD>
<TD> $13,000.00</TD>
</TR>
<TR>
<TD>Raphaely</TD>
<TD> 30</TD>
<TD> $11,000.00</TD>
</TR>
<TR>
<TD>Mavris</TD>
<TD> 40</TD>
<TD> $6,500.00</TD>
</TR>
<TR>
<TD>Weiss</TD>
<TD> 50</TD>
<TD> $8,000.00</TD>
</TR>
<TR>
<TD>Kaufling</TD>
<TD> 50</TD>
<TD> $7,900.00</TD>
</TR>
<TR>
<TD>Fripp</TD>
<TD> 50</TD>
<TD> $8,200.00</TD>
</TR>
<TR>
<TD>Vollman</TD>
<TD> 50</TD>
<TD> $6,500.00</TD>
</TR>
```

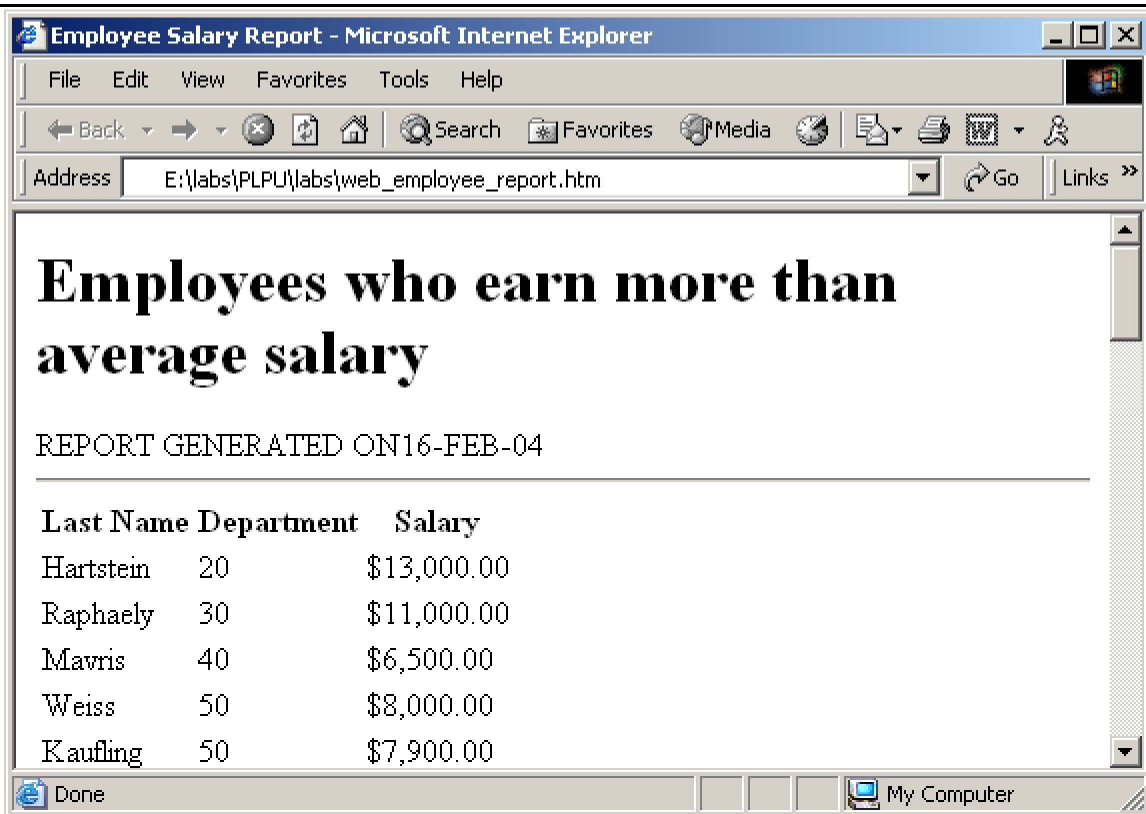
Practice 5: Solutions (continued)

```
<TR>
<TD>Hunold</TD>
<TD> 60</TD>
<TD> $9,000.00</TD>
</TR>
<TR>
<TD>Baer</TD>
<TD> 70</TD>
<TD> $10,000.00</TD>
</TR>
<TR> <TD>Russell</TD> <TD> 80</TD> <TD> $14,000.00</TD> </TR> <TR>
<TD>Bernstein</TD> <TD> 80</TD> <TD>
$9,500.00</TD> </TR> <TR> <TD>Olsen</TD> <TD> 80</TD> <TD> $8,000.00</TD>
</TR> <TR> <TD>Vishney</TD> <TD> 80</TD> <TD> $10,500.00</TD>
</TR> <TR> <TD>Sewall</TD> <TD> 80</TD> <TD> $7,000.00</TD> </TR> <TR>
<TD>Doran</TD> <TD> 80</TD> <TD>
$7,500.00</TD> </TR> <TR> <TD>Smith</TD> <TD> 80</TD> <TD> $8,000.00</TD>
</TR> <TR> <TD>McEwen</TD> <TD> 80</TD> <TD> $9,000.00</TD>
</TR> <TR> <TD>Sully</TD> <TD> 80</TD> <TD> $9,500.00</TD> </TR> <TR>
<TD>King</TD> <TD> 80</TD> <TD>
$10,000.00</TD> </TR> <TR> <TD>Tuvault</TD> <TD> 80</TD> <TD>
$7,000.00</TD> </TR> <TR> <TD>Cambrault</TD> <TD> 80</TD> <TD>
$7,500.00</TD>
</TR> <TR> <TD>Bates</TD> <TD> 80</TD> <TD> $7,300.00</TD> </TR> <TR>
<TD>Smith</TD> <TD> 80</TD> <TD>
$7,400.00</TD> </TR> <TR> <TD>Fox</TD> <TD> 80</TD> <TD> $9,600.00</TD>
</TR> <TR> <TD>Bloom</TD> <TD> 80</TD> <TD> $10,000.00</TD> </TR>
<TR> <TD>Ozer</TD> <TD> 80</TD> <TD> $11,500.00</TD> </TR> <TR>
<TD>Ande</TD> <TD> 80</TD> <TD> $6,400.00</TD> </TR> <TR> <TD>Lee</TD>
<TD>
80</TD> <TD> $6,800.00</TD> </TR> <TR> <TD>Marvins</TD> <TD> 80</TD> <TD>
$7,200.00</TD> </TR> <TR>
<TD>Greene</TD> <TD> 80</TD> <TD> $9,500.00</TD> </TR> <TR>
<TD>Livingston</TD> <TD> 80</TD> <TD> $8,400.00</TD> </TR> <TR>
<TD>Taylor</TD> <TD>
80</TD> <TD> $8,600.00</TD> </TR> <TR> <TD>Hutton</TD> <TD> 80</TD> <TD>
$8,800.00</TD> </TR>
<TR> <TD>Abel</TD> <TD> 80</TD> <TD> $11,000.00</TD> </TR> <TR>
<TD>Hall</TD> <TD> 80</TD> <TD> $9,000.00</TD> </TR> <TR> <TD>Tucker</TD>
<TD>
80</TD> <TD> $10,000.00</TD> </TR> <TR> <TD>Partners</TD> <TD> 80</TD>
<TD> $13,500.00</TD> </TR> <TR>
<TD>Errazuriz</TD> <TD> 80</TD> <TD> $12,000.00</TD> </TR> <TR>
<TD>Zlotkey</TD> <TD> 80</TD> <TD>
$10,500.00</TD> </TR> <TR> <TD>Cambrault</TD> <TD> 80</TD> <TD>
$11,000.00</TD> </TR> <TR> <TD>King</TD> <TD> 90</TD> <TD>
$24,000.00</TD> </TR>
<TR> <TD>Kochhar</TD> <TD> 90</TD> <TD> $17,000.00</TD> </TR> <TR> <TD>De
Haan</TD> <TD> 90</TD> <TD>
$17,000.00</TD> </TR> <TR> <TD>Greenberg</TD> <TD> 100</TD> <TD>
$12,000.00</TD> </TR>
```

Practice 5: Solutions (continued)

```
<TR> <TD>Faviet</TD> <TD> 100</TD> <TD> $9,000.00</TD>
</TR> <TR> <TD>Chen</TD> <TD> 100</TD> <TD> $8,200.00</TD> </TR> <TR>
<TD>Sciarra</TD> <TD> 100</TD> <TD>
$7,700.00</TD> </TR> <TR> <TD>Urman</TD> <TD> 100</TD> <TD>
$7,800.00</TD> </TR> <TR> <TD>Popp</TD> <TD> 100</TD> <TD> $6,900.00</TD>
</TR>
<TR> <TD>Higgins</TD> <TD> 110</TD> <TD> $12,000.00</TD> </TR> <TR>
<TD>Gietz</TD> <TD> 110</TD> <TD>
$8,300.00</TD> </TR> <TR> <TD>Grant</TD> <TD>-</TD> <TD> $7,000.00</TD>
</TR> </TABLE> <HR> *** END OF REPORT *** </BODY> </HTML>
PL/SQL procedure successfully completed.
```

d. Create an HTML file called `web_employee_report.htm` containing the output result text that you select and copy from the opening `<HTML>` tag to the closing `</HTML>` tag. Paste the copied text into the file and save it to disk. Double-click the file to display the results in your default browser.



Practice 5: Solutions (continued)

3. Your boss wants to run the employee report frequently. You create a procedure that uses the DBMS_SCHEDULER package to schedule the EMPLOYEE_REPORT procedure for execution. You should use parameters to specify a frequency, and an optional argument to specify the number of minutes after which the scheduled job should be terminated.

   a. Create a procedure called SCHEDULE_REPORT that provides the following two parameters:
      - interval to specify a string indicating the frequency of the scheduled job
      - minutes to specify the total life in minutes (default of 10) for the scheduled job, after which it is terminated. The code will divide the duration by the quantity (24×60) when it is added to the current date and time to specify the termination time.

      When the procedure creates a job, with the name of EMPSAL_REPORT by calling DBMS_SCHEDULER.CREATE_JOB, the job should be enabled and scheduled for the PL/SQL block to start immediately. You must schedule an anonymous block to invoke the EMPLOYEE_REPORT procedure so that the file name can be updated with a new time, each time the report is executed. EMPLOYEE_REPORT is given the directory name supplied by your instructor for task 1, and the file name parameter is specified in the following format:
      sal_rptxx_hh24-mi-ss.txt, where xx is your assigned user number and hh24-mi-ss represents the hours, minutes, and seconds

      Use the following local PL/SQL variable to construct a PL/SQL block:

```
plsql_block VARCHAR2(200) :=
 'BEGIN'||
 ' EMPLOYEE_REPORT(''UTL_FILE'','||
 '''sal_rptXX_''||to_char(sysdate,''HH24-MI-SS'')||''.txt'');'||
 'END;';
```

      This code is provided to help you because it is a nontrivial PL/SQL string to construct. In the PL/SQL block, **XX** is your student number.

```
CREATE OR REPLACE PROCEDURE schedule_report(
  interval VARCHAR2, minutes NUMBER := 10) IS
  plsql_block VARCHAR2(200) :=
    'BEGIN'||
    ' EMPLOYEE_REPORT(''UTL_FILE'','||
       '''sal_rpt01_''||to_char(sysdate,''HH24-MI-SS'')||''.txt''); '||
    'END;';
BEGIN
```

Practice 5: Solutions (continued)

```
  DBMS_SCHEDULER.CREATE_JOB(
     job_name => 'EMPSAL_REPORT',
     job_type => 'PLSQL_BLOCK',
     job_action => plsql_block,
     start_date => SYSDATE,
     repeat_interval => interval,
     end_date => SYSDATE + minutes/(24*60),
     enabled => TRUE);
END;
/
SHOW ERRORS

Procedure created.

No errors.
```

b. Test the `SCHEDULE_REPORT` procedure by executing it with a parameter specifying a frequency of every 2 minutes and a termination time 10 minutes after it starts.
Note: You will have to connect to the database server by using `PSFTP` to check whether your files are created.

```
EXECUTE schedule_report('FREQUENCY=MINUTELY;INTERVAL=2', 10)

PL/SQL procedure successfully completed.
```

c. During and after the process, you can query `job_name` and enabled columns from the `USER_SCHEDULER_JOBS` table to check whether the job still exists.

```
SELECT job_name, enabled
FROM user_scheduler_jobs;
```

Note: This query should return no rows after 10 minutes have elapsed.

Practice 6: Solutions

1. Create a package called TABLE_PKG that uses Native Dynamic SQL to create or drop a table, and to populate, modify, and delete rows from the table.

   a. Create a package specification with the following procedures:
   ```
   PROCEDURE make(table_name VARCHAR2, col_specs VARCHAR2)
   PROCEDURE add_row(table_name VARCHAR2, col_values VARCHAR2,
     cols VARCHAR2 := NULL)
   PROCEDURE upd_row(table_name VARCHAR2, set_values VARCHAR2,
     conditions VARCHAR2 := NULL)
   PROCEDURE del_row(table_name VARCHAR2,
     conditions VARCHAR2 := NULL)
   PROCEDURE remove(table_name VARCHAR2)
   ```
   Ensure that subprograms manage optional default parameters with NULL values.

```
CREATE OR REPLACE PACKAGE table_pkg IS
  PROCEDURE make(table_name VARCHAR2, col_specs VARCHAR2);
  PROCEDURE add_row(table_name VARCHAR2, col_values VARCHAR2,
    cols VARCHAR2 := NULL);
  PROCEDURE upd_row(table_name VARCHAR2, set_values VARCHAR2,
    conditions VARCHAR2 := NULL);
  PROCEDURE del_row(table_name VARCHAR2, conditions VARCHAR2 := NULL);
  PROCEDURE remove(table_name VARCHAR2);
END table_pkg;
/
SHOW ERRORS

Package created.

No errors.
```

   b. Create the package body that accepts the parameters and dynamically constructs the appropriate SQL statements that are executed using Native Dynamic SQL, except for the remove procedure that should be written using the DBMS_SQL package.

```
CREATE OR REPLACE PACKAGE BODY table_pkg IS
  PROCEDURE execute(stmt VARCHAR2) IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE(stmt);
    EXECUTE IMMEDIATE stmt;
  END;

  PROCEDURE make(table_name VARCHAR2, col_specs VARCHAR2) IS
    stmt VARCHAR2(200) := 'CREATE TABLE '|| table_name ||
                          ' (' || col_specs || ')';
  BEGIN
    execute(stmt);
  END;
```

Practice 6: Solutions (continued)

```
  PROCEDURE add_row(table_name VARCHAR2, col_values VARCHAR2,
    cols VARCHAR2 := NULL) IS
    stmt VARCHAR2(200) := 'INSERT INTO '|| table_name;
  BEGIN
    IF cols IS NOT NULL THEN
       stmt := stmt || ' (' || cols || ')';
    END IF;
    stmt := stmt || ' VALUES (' || col_values || ')';
    execute(stmt);
  END;

  PROCEDURE upd_row(table_name VARCHAR2, set_values VARCHAR2,
    conditions VARCHAR2 := NULL) IS
    stmt VARCHAR2(200) := 'UPDATE '|| table_name || ' SET ' ||
set_values;
  BEGIN
    IF conditions IS NOT NULL THEN
       stmt := stmt || ' WHERE ' || conditions;
    END IF;
    execute(stmt);
  END;

  PROCEDURE del_row(table_name VARCHAR2, conditions VARCHAR2 := NULL) IS
    stmt VARCHAR2(200) := 'DELETE FROM '|| table_name;
  BEGIN
    IF conditions IS NOT NULL THEN
       stmt := stmt || ' WHERE ' || conditions;
    END IF;
    execute(stmt);
  END;

  PROCEDURE remove(table_name VARCHAR2) IS
    csr_id INTEGER;
    stmt VARCHAR2(100) := 'DROP TABLE '||table_name;
  BEGIN
    csr_id := DBMS_SQL.OPEN_CURSOR;
    DBMS_OUTPUT.PUT_LINE(stmt);
    DBMS_SQL.PARSE(csr_id, stmt, DBMS_SQL.NATIVE);
    -- Parse executes DDL statements,no EXECUTE is required.
    DBMS_SQL.CLOSE_CURSOR(csr_id);
  END;

END table_pkg;
/
SHOW ERRORS

Package body created.

No errors.
```

c. Execute the package MAKE procedure to create a table as follows:

```
make('my_contacts', 'id number(4), name varchar2(40)');
```

```
EXECUTE table_pkg.make('my_contacts', 'id number(4), name varchar2(40)')

PL/SQL procedure successfully completed.
```

d. Describe the MY_CONTACTS table structure.

```
DESCRIBE my_contacts
```

Name	Null?	Type
ID		NUMBER(4)
NAME		VARCHAR2(40)

e. Execute the ADD_ROW package procedure to add the following rows:

```
add_row('my_contacts','1,''Geoff Gallus''','id, name');
add_row('my_contacts','2,''Nancy''','id, name');
add_row('my_contacts','3,''Sunitha Patel''','id,name');
add_row('my_contacts','4,''Valli Pataballa''','id,name');
```

```
BEGIN
  table_pkg.add_row('my_contacts','1,''Geoff Gallus''','id, name');
  table_pkg.add_row('my_contacts','2,''Nancy''','id, name');
  table_pkg.add_row('my_contacts','3,''Sunitha Patel''','id,name');
  table_pkg.add_row('my_contacts','4,''Valli Pataballa''','id,name');
END;
/

PL/SQL procedure successfully completed.
```

f. Query the MY_CONTACTS table contents.

```
SELECT *
FROM my_contacts;
```

ID	NAME
1	Geoff Gallus
2	Nancy
3	Sunitha Patel
4	Valli Pataballa

g. Execute the DEL_ROW package procedure to delete a contact with ID value 1.

```
EXECUTE table_pkg.del_row('my_contacts', 'id=1')

PL/SQL procedure successfully completed.
```

Practice 6: Solutions (continued)

h. Execute the UPD_ROW procedure with following row data:

```
upd_row('my_contacts','name=''Nancy Greenberg''','id=2');
```

```
EXEC table_pkg.upd_row('my_contacts','name=''Nancy Greenberg''','id=2')

PL/SQL procedure successfully completed.
```

i. Select the data from the MY_CONTACTS table again to view the changes.

```
SELECT *
FROM my_contacts;
```

ID	NAME
2	Nancy Greenberg
3	Sunitha Patel
4	Valli Pataballa

j. Drop the table by using the remove procedure and describe the MY_CONTACTS table.

```
EXECUTE table_pkg.remove('my_contacts')
DESCRIBE my_contacts

PL/SQL procedure successfully completed.

ERROR:
ORA-04043: object my_contacts does not exist
```

2. Create a COMPILE_PKG package that compiles the PL/SQL code in your schema.

a. In the specification, create a package procedure called MAKE that accepts the name of a PL/SQL program unit to be compiled.

```
CREATE OR REPLACE PACKAGE compile_pkg IS
  PROCEDURE make(name VARCHAR2);
END compile_pkg;
/
SHOW ERRORS

Package created.

No errors.
```

Practice 6: Solutions (continued)

b. In the body, the `MAKE` procedure should call a private function called `GET_TYPE` to determine the PL/SQL object type from the data dictionary, and return the type name (use `PACKAGE` for a package with a body) if the object exists; otherwise, it should return a `NULL`. If the object exists, `MAKE` dynamically compiles it with the `ALTER` statement.

```
CREATE OR REPLACE PACKAGE BODY compile_pkg IS
  PROCEDURE execute(stmt VARCHAR2) IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE(stmt);
    EXECUTE IMMEDIATE stmt;
  END;

  FUNCTION get_type(name VARCHAR2) RETURN VARCHAR2 IS
    proc_type VARCHAR2(30) := NULL;
  BEGIN
    /*
     * The ROWNUM = 1 is added to the condition
     * to ensure only one row is returned if the
     * name represents a PACKAGE, which may also
     * have a PACKAGE BODY. In this case, we can
     * only compile the complete package, but not
     * the specification or body as separate
     * components.
     */
    SELECT object_type INTO proc_type
    FROM user_objects
    WHERE object_name = UPPER(name)
    AND ROWNUM = 1;
    RETURN proc_type;
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
      RETURN NULL;
  END;

  PROCEDURE make(name VARCHAR2) IS
    stmt       VARCHAR2(100);
    proc_type  VARCHAR2(30) := get_type(name);
  BEGIN
    IF proc_type IS NOT NULL THEN
      stmt := 'ALTER '|| proc_type ||' '|| name ||' COMPILE';
      execute(stmt);
    ELSE
      RAISE_APPLICATION_ERROR(-20001,
         'Subprogram '''|| name ||''' does not exist');
    END IF;
  END make;
END compile_pkg;
/
SHOW ERRORS
```

```
Package body created.

No errors.
```

c. Use the COMPILE_PKG.MAKE procedure to compile the EMPLOYEE_REPORT procedure, the EMP_PKG package, and a nonexistent object called EMP_DATA.

```
EXECUTE compile_pkg.make('employee_report')
EXECUTE compile_pkg.make('emp_pkg')
EXECUTE compile_pkg.make('emp_data')

ALTER PROCEDURE employee_report COMPILE
PL/SQL procedure successfully completed.

ALTER PACKAGE emp_pkg COMPILE
PL/SQL procedure successfully completed

BEGIN compile_pkg.make('emp_data'); END;

*

ERROR at line 1:
ORA-20001: Subprogram 'emp_data' does not exist
ORA-06512: at "ORA1.COMPILE_PKG", line 39
ORA-06512: at line 1
```

3. Add a procedure to the COMPILE_PKG that uses the DBMS_METADATA to obtain a DDL statement that can regenerate a named PL/SQL subprogram, and writes the DDL to a file by using the UTL_FILE package.

   a. In the package specification, create a procedure called REGENERATE that accepts the name of a PL/SQL component to be regenerated. Declare a public VARCHAR2 variable called dir initialized with the directory alias value 'UTL_FILE'. Compile the specification.

```
CREATE OR REPLACE PACKAGE compile_pkg IS
  dir VARCHAR2(100) := 'UTL_FILE';
  PROCEDURE make(name VARCHAR2);
  PROCEDURE regenerate(name VARCHAR2);
END compile_pkg;
/
SHOW ERRORS

Package created.

No errors.
```

Note: Initialize the correct path name in the dir variable value for your course.

Practice 6: Solutions (continued)

b. In the package body, implement the REGENERATE procedure so that it uses the GET_TYPE function to determine the PL/SQL object type from the supplied name. If the object exists, then obtain the DDL used to create the component using the procedure DBMS_METADATA.GET_DDL, which must be provided with the object name in uppercase text.
Save the DDL statement in a file by using the UTL_FILE.PUT procedure. Write the file in the directory path stored in the public variable called dir (from the specification). Construct a file name (in lowercase characters) by concatenating the USER function, an underscore, and the object name with a .sql extension. For example: ora1_myobject.sql. Compile the body.

```
CREATE OR REPLACE PACKAGE BODY compile_pkg IS

  PROCEDURE execute(stmt VARCHAR2) IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE(stmt);
    EXECUTE IMMEDIATE stmt;
  END;

  FUNCTION get_type(name VARCHAR2) RETURN VARCHAR2 IS
    proc_type VARCHAR2(30) := NULL;
  BEGIN
    /*
     * The ROWNUM = 1 is added to the condition
     * to ensure only one row is returned if the
     * name represents a PACKAGE, which may also
     * have a PACKAGE BODY. In this case, we can
     * only compile the complete package, but not
     * the specification or body as separate
     * components.
     */
    SELECT object_type INTO proc_type
    FROM user_objects
    WHERE object_name = UPPER(name)
    AND ROWNUM = 1;
    RETURN proc_type;
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
      RETURN NULL;
  END;
```

Practice 6: Solutions (continued)

```
  PROCEDURE make(name VARCHAR2) IS
    stmt       VARCHAR2(100);
    proc_type  VARCHAR2(30) := get_type(name);
  BEGIN
    IF proc_type IS NOT NULL THEN
      stmt := 'ALTER '|| proc_type ||' '|| name ||' COMPILE';
      execute(stmt);
    ELSE
      RAISE_APPLICATION_ERROR(-20001,
         'Subprogram '''|| name ||''' does not exist');
    END IF;
  END make;

  PROCEDURE regenerate (name VARCHAR2) IS
    file UTL_FILE.FILE_TYPE;
    filename VARCHAR2(100) := LOWER(USER ||'_'|| name ||'.sql');
    proc_type  VARCHAR2(30) := get_type(name);
  BEGIN
    IF proc_type IS NOT NULL THEN
      file := UTL_FILE.FOPEN(dir, filename, 'w');
      UTL_FILE.PUT(file,
        DBMS_METADATA.GET_DDL(proc_type, UPPER(name)));
      UTL_FILE.FCLOSE(file);
    ELSE
      RAISE_APPLICATION_ERROR(-20001,
         'Object with '''|| name ||''' does not exist');
    END IF;
  END regenerate;

END compile_pkg;
/
SHOW ERRORS

Package body created.

No errors.
```

c. Execute the `COMPILE_PKG.REGENERATE` procedure by using the name of the `TABLE_PKG` created in the first task of this practice.

```
EXECUTE compile_pkg.regenerate('TABLE_PKG')
```

Note: If required, you can execute the following statement to set the directory for the file:

```
EXECUTE compile_pkg.dir := '<utl_file_dir>';
```

Practice 6: Solutions (continued)

d. Use Putty FTP to get the generated file from the server to your local directory. Edit the file to place a / terminator character at the end of a `CREATE` statement (if required). Cut and paste the results into the *i*SQL*Plus buffer and execute the statement.

Here is a sample Putty FTP session:

```
psftp> open esslin05
login as: teach7
Using username "teach7".
Password: ******
Remote working directory is /home1/teach7
psftp> cd UTL_FILE
Remote directory is now /home1/teach7/UTL_FILE
psftp> lcd E:\labs\PLPU\labs
New local directory is E:\labs\PLPU\labs
psftp> get ora1_emp_pkg.sql
remote:/home1/teach7/UTL_FILE/ora1_emp_pkg.sql => local:ora1_emp_pkg.sql
psftp> exit
```

Practice 7: Solutions

1. Update EMP_PKG with a new procedure to query employees in a specified department.

   a. In the specification, declare a get_employees procedure, with its parameter called dept_id based on the employees.department_id column type. Define an index-by PL/SQL type as a TABLE OF EMPLOYEES%ROWTYPE.

```
CREATE OR REPLACE PACKAGE emp_pkg IS
  TYPE emp_tabtype IS TABLE OF employees%ROWTYPE;
  PROCEDURE add_employee(
    first_name employees.first_name%TYPE,
    last_name employees.last_name%TYPE,
    email employees.email%TYPE,
    job employees.job_id%TYPE DEFAULT 'SA_REP',
    mgr employees.manager_id%TYPE DEFAULT 145,
    sal employees.salary%TYPE DEFAULT 1000,
    comm employees.commission_pct%TYPE DEFAULT 0,
    deptid employees.department_id%TYPE DEFAULT 30);
  PROCEDURE add_employee(
    first_name employees.first_name%TYPE,
    last_name employees.last_name%TYPE,
    deptid employees.department_id%TYPE);
  PROCEDURE get_employee(
    empid IN employees.employee_id%TYPE,
    sal OUT employees.salary%TYPE,
    job OUT employees.job_id%TYPE);
  FUNCTION get_employee(emp_id employees.employee_id%type)
    return employees%rowtype;
  FUNCTION get_employee(family_name employees.last_name%type)
    return employees%rowtype;
  PROCEDURE get_employees(dept_id employees.department_id%type);
  PROCEDURE init_departments;
  PROCEDURE print_employee(emprec employees%rowtype);
END emp_pkg;
/
SHOW ERRORS

Package created.

No errors.
```

   b. In the body of the package, define a private variable called emp_table based on the type defined in the specification to hold employee records. Implement the get_employees procedure to bulk fetch the data into the table.

```
CREATE OR REPLACE PACKAGE BODY emp_pkg IS
  TYPE boolean_tabtype IS TABLE OF BOOLEAN
    INDEX BY BINARY_INTEGER;
  valid_departments boolean_tabtype;
  emp_table         emp_tabtype;
```

Practice 7: Solutions (continued)

```
FUNCTION valid_deptid(deptid IN departments.department_id%TYPE)
 RETURN BOOLEAN;

PROCEDURE add_employee(
  first_name employees.first_name%TYPE,
  last_name employees.last_name%TYPE,
  email employees.email%TYPE,
  job employees.job_id%TYPE DEFAULT 'SA_REP',
  mgr employees.manager_id%TYPE DEFAULT 145,
  sal employees.salary%TYPE DEFAULT 1000,
  comm employees.commission_pct%TYPE DEFAULT 0,
  deptid employees.department_id%TYPE DEFAULT 30) IS
BEGIN
  IF valid_deptid(deptid) THEN
    INSERT INTO employees(employee_id, first_name, last_name, email,
      job_id,manager_id,hire_date,salary,commission_pct,department_id)
    VALUES (employees_seq.NEXTVAL, first_name, last_name, email,
      job, mgr, TRUNC(SYSDATE), sal, comm, deptid);
  ELSE
    RAISE_APPLICATION_ERROR (-20204,
                            'Invalid department ID. Try again.');
  END IF;
END add_employee;

PROCEDURE add_employee(
  first_name employees.first_name%TYPE,
  last_name employees.last_name%TYPE,
  deptid employees.department_id%TYPE) IS
  email employees.email%type;
BEGIN
  email := UPPER(SUBSTR(first_name, 1, 1)||SUBSTR(last_name, 1, 7));
  add_employee(first_name, last_name, email, deptid => deptid);
END;

PROCEDURE get_employee(
  empid IN employees.employee_id%TYPE,
  sal OUT employees.salary%TYPE,
  job OUT employees.job_id%TYPE) IS
BEGIN
  SELECT salary, job_id
  INTO sal, job
  FROM employees
  WHERE employee_id = empid;
END get_employee;
```

Practice 7: Solutions (continued)

```
FUNCTION get_employee(emp_id employees.employee_id%type)
  return employees%rowtype IS
  emprec employees%rowtype;
BEGIN
  SELECT * INTO emprec
  FROM employees
  WHERE employee_id = emp_id;
  RETURN emprec;
END;

FUNCTION get_employee(family_name employees.last_name%type)
  return employees%rowtype IS
  emprec employees%rowtype;
BEGIN
  SELECT * INTO emprec
  FROM employees
  WHERE last_name = family_name;
  RETURN emprec;
END;

PROCEDURE get_employees(dept_id employees.department_id%type) IS
BEGIN
  SELECT * BULK COLLECT INTO emp_table
  FROM EMPLOYEES
  WHERE department_id = dept_id;
END;

PROCEDURE init_departments IS
BEGIN
  FOR rec IN (SELECT department_id FROM departments)
  LOOP
    valid_departments(rec.department_id) := TRUE;
  END LOOP;
END;

PROCEDURE print_employee(emprec employees%rowtype) IS
BEGIN
  DBMS_OUTPUT.PUT_LINE(emprec.department_id ||' '||
                       emprec.employee_id||' '||
                       emprec.first_name||' '||
                       emprec.last_name||' '||
                       emprec.job_id||' '||
                       emprec.salary);
END;
```

Practice 7: Solutions (continued)

```
  FUNCTION valid_deptid(deptid IN departments.department_id%TYPE)
   RETURN BOOLEAN IS
    dummy PLS_INTEGER;
  BEGIN
    RETURN valid_departments.exists(deptid);
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
    RETURN FALSE;
  END valid_deptid;

BEGIN
  init_departments;
END emp_pkg;
/
SHOW ERRORS

Package body created.

No errors.
```

c. Create a new procedure in the specification and body, called `show_employees`, which does not take arguments and displays the contents of the private PL/SQL table variable (if any data exists).
Hint: Use the `print_employee` procedure.

```
CREATE OR REPLACE PACKAGE emp_pkg IS
  TYPE emp_tabtype IS TABLE OF employees%ROWTYPE;
  PROCEDURE add_employee(
    first_name employees.first_name%TYPE,
    last_name employees.last_name%TYPE,
    email employees.email%TYPE,
    job employees.job_id%TYPE DEFAULT 'SA_REP',
    mgr employees.manager_id%TYPE DEFAULT 145,
    sal employees.salary%TYPE DEFAULT 1000,
    comm employees.commission_pct%TYPE DEFAULT 0,
    deptid employees.department_id%TYPE DEFAULT 30);
  PROCEDURE add_employee(
    first_name employees.first_name%TYPE,
    last_name employees.last_name%TYPE,
    deptid employees.department_id%TYPE);
  PROCEDURE get_employee(
    empid IN employees.employee_id%TYPE,
    sal OUT employees.salary%TYPE,
    job OUT employees.job_id%TYPE);
  FUNCTION get_employee(emp_id employees.employee_id%type)
    return employees%rowtype;
  FUNCTION get_employee(family_name employees.last_name%type)
    return employees%rowtype;
  PROCEDURE get_employees(dept_id employees.department_id%type);
  PROCEDURE init_departments;
```

Practice 7: Solutions (continued)

```
  PROCEDURE print_employee(emprec employees%rowtype);
  PROCEDURE show_employees;
END emp_pkg;
/
SHOW ERRORS

CREATE OR REPLACE PACKAGE BODY emp_pkg IS
  TYPE boolean_tabtype IS TABLE OF BOOLEAN
    INDEX BY BINARY_INTEGER;
  valid_departments boolean_tabtype;
  emp_table         emp_tabtype;

  FUNCTION valid_deptid(deptid IN departments.department_id%TYPE)
   RETURN BOOLEAN;

  PROCEDURE add_employee(
    first_name employees.first_name%TYPE,
    last_name employees.last_name%TYPE,
    email employees.email%TYPE,
    job employees.job_id%TYPE DEFAULT 'SA_REP',
    mgr employees.manager_id%TYPE DEFAULT 145,
    sal employees.salary%TYPE DEFAULT 1000,
    comm employees.commission_pct%TYPE DEFAULT 0,
    deptid employees.department_id%TYPE DEFAULT 30) IS
  BEGIN
    IF valid_deptid(deptid) THEN
      INSERT INTO employees(employee_id, first_name, last_name, email,
        job_id,manager_id,hire_date,salary,commission_pct,department_id)
      VALUES (employees_seq.NEXTVAL, first_name, last_name, email,
        job, mgr, TRUNC(SYSDATE), sal, comm, deptid);
    ELSE
      RAISE_APPLICATION_ERROR (-20204,
                               'Invalid department ID. Try again.');
    END IF;
  END add_employee;

  PROCEDURE add_employee(
    first_name employees.first_name%TYPE,
    last_name employees.last_name%TYPE,
    deptid employees.department_id%TYPE) IS
    email employees.email%type;
  BEGIN
    email := UPPER(SUBSTR(first_name, 1, 1)||SUBSTR(last_name, 1, 7));
    add_employee(first_name, last_name, email, deptid => deptid);
  END;
```

Practice 7: Solutions (continued)

```
PROCEDURE get_employee(
  empid IN employees.employee_id%TYPE,
  sal OUT employees.salary%TYPE,
  job OUT employees.job_id%TYPE) IS
BEGIN
  SELECT salary, job_id
  INTO sal, job
  FROM employees
  WHERE employee_id = empid;
END get_employee;

FUNCTION get_employee(emp_id employees.employee_id%type)
  return employees%rowtype IS
  emprec employees%rowtype;
BEGIN
  SELECT * INTO emprec
  FROM employees
  WHERE employee_id = emp_id;
  RETURN emprec;
END;

FUNCTION get_employee(family_name employees.last_name%type)
  return employees%rowtype IS
  emprec employees%rowtype;
BEGIN
  SELECT * INTO emprec
  FROM employees
  WHERE last_name = family_name;
  RETURN emprec;
END;

PROCEDURE get_employees(dept_id employees.department_id%type) IS
BEGIN
  SELECT * BULK COLLECT INTO emp_table
  FROM EMPLOYEES
  WHERE department_id = dept_id;
END;

PROCEDURE init_departments IS
BEGIN
  FOR rec IN (SELECT department_id FROM departments)
  LOOP
    valid_departments(rec.department_id) := TRUE;
  END LOOP;
END;
```

Practice 7: Solutions (continued)

```
  PROCEDURE print_employee(emprec employees%rowtype) IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE(emprec.department_id ||' '||
                         emprec.employee_id||' '||
                         emprec.first_name||' '||
                         emprec.last_name||' '||
                         emprec.job_id||' '||
                         emprec.salary);
  END;

  PROCEDURE show_employees IS
  BEGIN
    IF emp_table IS NOT NULL THEN
      DBMS_OUTPUT.PUT_LINE('Employees in Package table');
      FOR i IN 1 .. emp_table.COUNT
      LOOP
        print_employee(emp_table(i));
      END LOOP;
    END IF;
  END show_employees;

  FUNCTION valid_deptid(deptid IN departments.department_id%TYPE)
   RETURN BOOLEAN IS
    dummy PLS_INTEGER;
  BEGIN
    RETURN valid_departments.exists(deptid);
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
    RETURN FALSE;
  END valid_deptid;

BEGIN
  init_departments;
END emp_pkg;
/
SHOW ERRORS

Package created.

No errors.

Package body created.

No errors.
```

d. Invoke the `emp_pkg.get_employees` procedure for department 30, and then invoke `emp_pkg.show_employees`. Repeat this for department 60.

```
EXECUTE emp_pkg.get_employees(30)
EXECUTE emp_pkg.show_employees

PL/SQL procedure successfully completed.

Employees in Package table
30 114 Den Raphaely PU_MAN 11000
30 115 Alexander Khoo PU_CLERK 3100
30 116 Shelli Baida PU_CLERK 2900
30 117 Sigal Tobias PU_CLERK 2800
30 118 Guy Himuro PU_CLERK 2600
30 119 Karen Colmenares PU_CLERK 2500
30 209 Samuel Joplin SA_REP 1000
PL/SQL procedure successfully completed.


EXECUTE emp_pkg.get_employees(60)
EXECUTE emp_pkg.show_employees

PL/SQL procedure successfully completed.

Employees in Package table
60 103 Alexander Hunold IT_PROG 9000
60 104 Bruce Ernst IT_PROG 6000
60 105 David Austin IT_PROG 4800
60 106 Valli Pataballa IT_PROG 4800
60 107 Diana Lorentz IT_PROG 4200
PL/SQL procedure successfully completed.
```

2. Your manager wants to keep a log whenever the `add_employee` procedure in the package is invoked to insert a new employee into the `EMPLOYEES` table.

   a. First, load and execute the `E:\labs\PLPU\labs\lab_07_02_a.sql` script to create a log table called `LOG_NEWEMP`, and a sequence called `log_newemp_seq`.

```
CREATE TABLE log_newemp (
  entry_id  NUMBER(6) CONSTRAINT log_newemp_pk PRIMARY KEY,
  user_id   VARCHAR2(30),
  log_time  DATE,
  name      VARCHAR2(60)
);

CREATE SEQUENCE log_newemp_seq;

Table created.

Sequence created.
```

b. In the package body, modify the `add_employee` procedure, which performs the actual `INSERT` operation to have a local procedure called `audit_newemp`. The `audit_newemp` procedure must use an autonomous transaction to insert a log record into the `LOG_NEWEMP` table. Store the `USER`, the current time, and the new employee name in the log table row. Use `log_newemp_seq` to set the `entry_id` column. **Note:** Remember to perform a `COMMIT` operation in a procedure with an autonomous transaction.

```
CREATE OR REPLACE PACKAGE BODY emp_pkg IS
  TYPE boolean_tabtype IS TABLE OF BOOLEAN
    INDEX BY BINARY_INTEGER;
  valid_departments boolean_tabtype;
  emp_table         emp_tabtype;

  FUNCTION valid_deptid(deptid IN departments.department_id%TYPE)
   RETURN BOOLEAN;

  PROCEDURE add_employee(
    first_name employees.first_name%TYPE,
    last_name employees.last_name%TYPE,
    email employees.email%TYPE,
    job employees.job_id%TYPE DEFAULT 'SA_REP',
    mgr employees.manager_id%TYPE DEFAULT 145,
    sal employees.salary%TYPE DEFAULT 1000,
    comm employees.commission_pct%TYPE DEFAULT 0,
    deptid employees.department_id%TYPE DEFAULT 30) IS

    PROCEDURE audit_newemp IS
      PRAGMA AUTONOMOUS_TRANSACTION;
      user_id VARCHAR2(30) := USER;
    BEGIN
      INSERT INTO log_newemp (entry_id, user_id, log_time, name)
      VALUES (log_newemp_seq.NEXTVAL, user_id, sysdate,
              first_name||' '||last_name);
      COMMIT;
    END audit_newemp;

  BEGIN
    IF valid_deptid(deptid) THEN
      INSERT INTO employees(employee_id, first_name, last_name, email,
        job_id,manager_id,hire_date,salary,commission_pct,department_id)
      VALUES (employees_seq.NEXTVAL, first_name, last_name, email,
        job, mgr, TRUNC(SYSDATE), sal, comm, deptid);
    ELSE
      RAISE_APPLICATION_ERROR (-20204,
                              'Invalid department ID. Try again.');
    END IF;
  END add_employee;
```

Practice 7: Solutions (continued)

```
  PROCEDURE add_employee(
    first_name employees.first_name%TYPE,
    last_name employees.last_name%TYPE,
    deptid employees.department_id%TYPE) IS
    email employees.email%type;
  BEGIN
    email := UPPER(SUBSTR(first_name, 1, 1)||SUBSTR(last_name, 1, 7));
    add_employee(first_name, last_name, email, deptid => deptid);
  END;

  ...

  FUNCTION valid_deptid(deptid IN departments.department_id%TYPE)
   RETURN BOOLEAN IS
    dummy PLS_INTEGER;
  BEGIN
    RETURN valid_departments.exists(deptid);
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
    RETURN FALSE;
  END valid_deptid;

BEGIN
  init_departments;
END emp_pkg;
/
SHOW ERRORS

Package body created.

No errors.
```

c. Modify the `add_employee` procedure to invoke `audit_emp` before it performs the insert operation.

```
CREATE OR REPLACE PACKAGE BODY emp_pkg IS
  TYPE boolean_tabtype IS TABLE OF BOOLEAN
    INDEX BY BINARY_INTEGER;
  valid_departments boolean_tabtype;
  emp_table         emp_tabtype;

  FUNCTION valid_deptid(deptid IN departments.department_id%TYPE)
   RETURN BOOLEAN;
```

Practice 7: Solutions (continued)

```
  PROCEDURE add_employee(
    first_name employees.first_name%TYPE,
    last_name employees.last_name%TYPE,
    email employees.email%TYPE,
    job employees.job_id%TYPE DEFAULT 'SA_REP',
    mgr employees.manager_id%TYPE DEFAULT 145,
    sal employees.salary%TYPE DEFAULT 1000,
    comm employees.commission_pct%TYPE DEFAULT 0,
    deptid employees.department_id%TYPE DEFAULT 30) IS

    PROCEDURE audit_newemp IS
      PRAGMA AUTONOMOUS_TRANSACTION;
      user_id VARCHAR2(30) := USER;
    BEGIN
      INSERT INTO log_newemp (entry_id, user_id, log_time, name)
      VALUES (log_newemp_seq.NEXTVAL, user_id, sysdate,
              first_name||' '||last_name);
      COMMIT;
    END audit_newemp;
  BEGIN
    IF valid_deptid(deptid) THEN
      audit_newemp;
      INSERT INTO employees(employee_id, first_name, last_name, email,
        job_id,manager_id,hire_date,salary,commission_pct,department_id)
      VALUES (employees_seq.NEXTVAL, first_name, last_name, email,
        job, mgr, TRUNC(SYSDATE), sal, comm, deptid);
    ELSE
      RAISE_APPLICATION_ERROR (-20204,
                               'Invalid department ID. Try again.');
    END IF;
  END add_employee;
  ...
  FUNCTION valid_deptid(deptid IN departments.department_id%TYPE)
   RETURN BOOLEAN IS
    dummy PLS_INTEGER;
  BEGIN
    RETURN valid_departments.exists(deptid);
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
    RETURN FALSE;
  END valid_deptid;

BEGIN
  init_departments;
END emp_pkg;
/
SHOW ERRORS

Package body created.

No errors.
```

Practice 7: Solutions (continued)

d. Invoke the `add_employee` procedure for these new employees: `Max Smart` in department 20 and `Clark Kent` in department 10. What happens?

```
EXECUTE emp_pkg.add_employee('Max', 'Smart', 20)
EXECUTE emp_pkg.add_employee('Clark', 'Kent', 10)

PL/SQL procedure successfully completed.

PL/SQL procedure successfully completed.
```

Both insert operations complete successfully, and the log table has two log records, as shown in the next step.

e. Query the two `EMPLOYEES` records added, and the records in the `LOG_NEWEMP` table. How many log records are present?

```
SELECT department_id, first_name, last_name
FROM employees
WHERE last_name IN ('Smart','Kent');
```

DEPARTMENT_ID	EMPLOYEE_ID	LAST_NAME	FIRST_NAME
10	222	Kent	Clark
20	221	Smart	Max

```
SELECT *
FROM log_newemp;
```

ENTRY_ID	USER_ID	LOG_TIME	NAME
1	ORA1	18-FEB-04	Max Smart
2	ORA1	18-FEB-04	Clark Kent

There are two log records, one for `Smart` and the other for `Kent`.

f. Execute a `ROLLBACK` statement to undo the insert operations that have not been committed. Use the same queries from Exercise 2e: the first to check whether the employee rows for `Smart` and `Kent` have been removed, and the second to check the log records in the `LOG_NEWEMP` table. How many log records are present? Why?

```
ROLLBACK;

Rollback complete.
```

Practice 7: Solutions (continued)

```
SELECT department_id, first_name, last_name
FROM employees
WHERE last_name IN ('Smart','Kent');

no rows selected

SELECT *
FROM log_newemp;
```

ENTRY_ID	USER_ID	LOG_TIME	NAME
1	ORA1	18-FEB-04	Max Smart
2	ORA1	18-FEB-04	Clark Kent

The two employee records are removed (rolled back). The two log records remain in the log table because they were inserted using an autonomous transaction, which is unaffected by the rollback performed in the main transaction.

If you have time, complete the following exercise:

3. Modify the `EMP_PKG` package to use `AUTHID` of `CURRENT_USER` and test the behavior with any other student.
   Note: Verify that the `LOG_NEWEMP` table exists from Exercise 2 in this practice.

   a. First, grant the `EXECUTE` privilege on your `EMP_PKG` package to another student.

```
Assume you are ORA1 and the other student is ORA2. You enter:
GRANT EXECUTE ON EMP_PKG TO ORA2;

Grant succeeded.
```

   b. Ask the other student to invoke your `add_employee` procedure to insert the employee `Jaco Pastorius` in department 10. Remember to prefix the package name with the owner of the package. The call should operate with definer's rights.

```
User ORA2 enters:
EXECUTE ora1.emp_pkg.add_employee('Jaco', 'Pastorius', 10)

PL/SQL procedure successfully completed.
```

c. Now, execute a query of the employees in department 10. In which user's employee table did the new record get inserted?

```
User ORA1 executes:
SELECT department_id, first_name, last_name
FROM employees
WHERE department_id = 10;
```

DEPARTMENT_ID	FIRST_NAME	LAST_NAME
10	Jennifer	Whalen
10	Jaco	Pastorius

```
User ORA2 executes:
SELECT department_id, first_name, last_name
FROM departments
WHERE department_id = 10;
```

DEPARTMENT_ID	FIRST_NAME	LAST_NAME
10	Jennifer	Whalen

The new employee is added to the table in the `ORA1` schema—that is, in the table of the owner of the `EMP_PKG` package.

d. Now, modify your package `EMP_PKG` specification to use an `AUTHID CURRENT_USER`. Compile the body of `EMP_PKG`.

```
User ORA1 executes:
CREATE OR REPLACE PACKAGE emp_pkg AUTHID CURRENT_USER IS
  TYPE emp_tabtype IS TABLE OF employees%ROWTYPE;
  PROCEDURE add_employee(
    first_name employees.first_name%TYPE,
    last_name employees.last_name%TYPE,
    email employees.email%TYPE,
    job employees.job_id%TYPE DEFAULT 'SA_REP',
    mgr employees.manager_id%TYPE DEFAULT 145,
    sal employees.salary%TYPE DEFAULT 1000,
    comm employees.commission_pct%TYPE DEFAULT 0,
    deptid employees.department_id%TYPE DEFAULT 30);
  PROCEDURE add_employee(
    first_name employees.first_name%TYPE,
    last_name employees.last_name%TYPE,
    deptid employees.department_id%TYPE);
  PROCEDURE get_employee(
    empid IN employees.employee_id%TYPE,
    sal OUT employees.salary%TYPE,
    job OUT employees.job_id%TYPE);
  FUNCTION get_employee(emp_id employees.employee_id%type)
    return employees%rowtype;
```

Practice 7: Solutions (continued)

```
  FUNCTION get_employee(family_name employees.last_name%type)
    return employees%rowtype;
  PROCEDURE get_employees(dept_id employees.department_id%type);
  PROCEDURE init_departments;
  PROCEDURE print_employee(emprec employees%rowtype);
  PROCEDURE show_employees;
END emp_pkg;
/
SHOW ERRORS

ALTER PACKAGE emp_pkg COMPILE BODY;

Package created.

No errors.

Package body altered.
```

e. Ask the same student to execute the `add_employee` procedure again to add employee `Joe Zawinal` in department 10.

Note: Make sure that the user `ORA2` has executed the Practice 7-2a and created the `log_newemp` table before executing `emp_pkg.add_employee`.

```
User ORA2 executes:

EXECUTE ora1.emp_pkg.add_employee('Joe', 'Zawinal', 10)

PL/SQL procedure successfully completed.
```

f. Query your employees in department 10. In which table was the new employee added?

```
User ORA1 executes:
SELECT department_id, first_name, last_name
FROM employees
WHERE department_id = 10;
```

DEPARTMENT_ID	FIRST_NAME	LAST_NAME
10	Jennifer	Whalen
10	Jaco	Pastorius

```
User ORA2 executes:
SELECT department_id, first_name, last_name
FROM employees
WHERE department_id = 10;
```

DEPARTMENT_ID	FIRST_NAME	LAST_NAME
10	Joe	Zawinal
10	Jennifer	Whalen

Practice 7: Solutions (continued)

The new employee is added to the user `ORA2` employee table. That is, the new employee is added to the table that is owned by the caller (invoker's rights) of the package procedure.

Practice 7: Solutions (continued)

g. Write a query to display the records added in the `LOG_NEWEMP` tables. Ask the other student to query his or her own copy of the table.

User ORA1 executes:

```
SELECT *
FROM log_newemp;
```

ENTRY_ID	USER_ID	LOG_TIME	NAME
1	ORA1	18-FEB-04	Max Smart
2	ORA1	18-FEB-04	Clark Kent
3	ORA2	18-FEB-04	Jaco Pastorius

User ORA2 executes:

```
SELECT *
FROM log_newemp;
```

ENTRY_ID	USER_ID	LOG_TIME	NAME
3	ORA2	18-FEB-04	Joe Zawinal
1	ORA2	18-FEB-04	Max Smart
2	ORA2	18-FEB-04	Clark Kent

The log records created by the `audit_emp` procedure (which executes the autonomous transaction) are stored in the log table of the owner of the package when the package procedure is executed with the definer's (owner) rights. The log records are stored in the caller's log table when the package procedure is executed with invoker's (caller) rights.

Practice 8: Solutions

1. Answer the following questions.

   a. Can a table or a synonym be invalidated?

      A table or a synonym can never be invalidated; however, dependent objects can be invalidated.

   b. Consider the following dependency example:

      ```
      The stand-alone procedure MY_PROC depends on the MY_PROC_PACK
      package procedure. The MY_PROC_PACK procedure's definition is
      changed by recompiling the package body. The MY_PROC_PACK
      procedure's declaration is not altered in the package
      specification.
      ```

      In this scenario, is the stand-alone procedure `MY_PROC` invalidated?

      No, it is not invalidated because the stand-alone procedure `MY_PROC` depends on the `MY_PROC_PACK` package procedure, which has not been altered. Although the package body is recompiled, the package specification is not invalidated and does not need to be recomplied.

2. Create a tree structure showing all dependencies involving your `add_employee` procedure and your `valid_deptid` function.
   Note: `add_employee` and `valid_deptid` were created in the lesson titled "Creating Stored Functions." You can run the solution scripts for Practice 2 if you need to create the procedure and function.

   a. Load and execute the `utldtree.sql` script, which is located in the `E:\lab\PLPU\Labs` folder.

      When you execute the script, the following results are displayed (you can ignore the error messages):

```
drop sequence deptree_seq
              *

ERROR at line 1:
ORA-02289: sequence does not exist
Sequence created.


drop table deptree_temptab
           *

ERROR at line 1:
ORA-00942: table or view does not exist
Table created.

Procedure created.
```

```
drop view deptree
*

ERROR at line 1:
ORA-00942: table or view does not exist

REM This view will succeed if current user is sys. This view shows
REM which shared cursors depend on the given object. If the current
REM user is not sys, then this view get an error either about lack
REM of privileges or about the non-existence of table x$kglxs.

set echo off

  from deptree_temptab d, dba_objects o
       *

ERROR at line 5:
ORA-00942: table or view does not exist

REM This view will succeed if current user is not sys. This view
REM does *not* show which shared cursors depend on the given object.
REM If the current user is sys then this view will get an error
REM indicating that the view already exists (since prior view create
REM will have succeeded).

set echo off
View created.


drop view ideptree
*

ERROR at line 1:
ORA-00942: table or view does not exist
View created.
```

b. Execute the `deptree_fill` procedure for the `add_employee` procedure.

```
EXECUTE deptree_fill('PROCEDURE', USER, 'add_employee')

PL/SQL procedure successfully completed.
```

c. Query the `IDEPTREE` view to see your results.

```
SELECT * FROM IDEPTREE;
```

DEPENDENCIES
PROCEDURE ORA1.ADD_EMPLOYEE

Practice 8: Solutions (continued)

d. Execute the `deptree_fill` procedure for the `valid_deptid` function.

```
EXECUTE deptree_fill('FUNCTION', USER, 'valid_deptid')

PL/SQL procedure successfully completed.
```

e. Query the `IDEPTREE` view to see your results.

```
SELECT * FROM IDEPTREE;
```

DEPENDENCIES
FUNCTION ORA1.VALID_DEPTID
PROCEDURE ORA1.ADD_EMPLOYEE

If you have time, complete the following exercise:

3. Dynamically validate invalid objects.

a. Make a copy of your `EMPLOYEES` table, called `EMPS`.

```
CREATE TABLE emps AS
  SELECT * FROM employees;

Table created.
```

b. Alter your `EMPLOYEES` table and add the `TOTSAL` column with the `NUMBER(9,2)` data type.

```
ALTER TABLE employees
  ADD (totsal NUMBER(9,2));

Table altered.
```

c. Create and save a query (`lab8_soln_3c.sql`) to display the name, type, and status of all invalid objects.

```
SELECT object_name, object_type, status
FROM USER_OBJECTS
WHERE status = 'INVALID';
```

Practice 8: Solutions (continued)

OBJECT_NAME	OBJECT_TYPE	STATUS
EMP_DETAILS_VIEW	VIEW	INVALID
SECURE_EMPLOYEES	TRIGGER	INVALID
UPDATE_JOB_HISTORY	TRIGGER	INVALID
TOTAL_SALARY	FUNCTION	INVALID
GET_EMPLOYEE	PROCEDURE	INVALID
GET_ANNUAL_COMP	FUNCTION	INVALID
ADD_EMPLOYEE	PROCEDURE	INVALID
EMP_PKG	PACKAGE	INVALID
EMP_PKG	PACKAGE BODY	INVALID
EMPLOYEE_REPORT	PROCEDURE	INVALID
WEB_EMPLOYEE_REPORT	PROCEDURE	INVALID

11 rows selected.

d. In `compile_pkg` (created in Practice 6 in the lesson titled "Dynamic SQL and Metadata"), add a procedure called `recompile` that recompiles all invalid procedures, functions, and packages in your schema. Use Native Dynamic SQL to `ALTER` the invalid object type and `COMPILE` it.

```
CREATE OR REPLACE PACKAGE compile_pkg IS
  PROCEDURE make(name VARCHAR2);
  PROCEDURE recompile;
END compile_pkg;
/
SHOW ERRORS

CREATE OR REPLACE PACKAGE BODY compile_pkg IS

  PROCEDURE execute(stmt VARCHAR2) IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE(stmt);
    EXECUTE IMMEDIATE stmt;
  END;
```

Practice 8: Solutions (continued)

```
  FUNCTION get_type(name VARCHAR2) RETURN VARCHAR2 IS
    proc_type VARCHAR2(30) := NULL;
  BEGIN
    /*
     * The ROWNUM = 1 is added to the condition
     * to ensure only one row is returned if the
     * name represents a PACKAGE, which may also
     * have a PACKAGE BODY. In this case, we can
     * only compile the complete package, but not
     * the specification or body as separate
     * components.
     */
    SELECT object_type INTO proc_type
    FROM user_objects
    WHERE object_name = UPPER(name)
    AND ROWNUM = 1;
    RETURN proc_type;
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
      RETURN NULL;
  END;

  PROCEDURE make(name VARCHAR2) IS
    stmt       VARCHAR2(100);
    proc_type  VARCHAR2(30) := get_type(name);
  BEGIN
    IF proc_type IS NOT NULL THEN
      stmt := 'ALTER '|| proc_type ||' '|| name ||' COMPILE';
      execute(stmt);
    ELSE
      RAISE_APPLICATION_ERROR(-20001,
         'Subprogram '''|| name ||''' does not exist');
    END IF;
  END make;

  PROCEDURE recompile IS
    stmt VARCHAR2(200);
    obj_name user_objects.object_name%type;
    obj_type user_objects.object_type%type;
  BEGIN
    FOR objrec IN (SELECT object_name, object_type
                   FROM user_objects
                   WHERE status = 'INVALID'
                   AND object_type <> 'PACKAGE BODY')
    LOOP
      stmt := 'ALTER '|| objrec.object_type ||' '||
                   objrec.object_name ||' COMPILE';
      execute(stmt);
    END LOOP;
  END recompile;
END compile_pkg;
/
```

Practice 8: Solutions (continued)

```
SHOW ERRORS

Package created.

No errors.

Package body created.

No errors.
```

e. Execute the `compile_pkg.recompile` procedure.

```
EXECUTE compile_pkg.recompile

PL/SQL procedure successfully completed.
```

f. Run the script file that you created in step 3c (`lab8_soln_3c.sql`) to check the status column value. Do you still have objects with an `INVALID` status?

```
SELECT object_name, object_type, status
FROM USER_OBJECTS
WHERE status = 'INVALID';

no rows selected
```

No rows are returned. There are no objects with an `INVALID` status.

Practice 9: Solutions

1. Create a table called PERSONNEL by executing the E:\labs\PLPU\labs\lab_09_01.sql script. The table contains the following attributes and data types:

Column Name	Data Type	Length
ID	NUMBER	6
last_name	VARCHAR2	35
review	CLOB	N/A
picture	BLOB	N/A

```
CREATE TABLE personnel (
 id        NUMBER(6) constraint personnel_id_pk PRIMARY KEY,
 last_name VARCHAR2(35),
 review   CLOB,
 picture  BLOB);

Table created.
```

2. Insert two rows into the PERSONNEL table, one each for employee 2034 (whose last name is Allen) and for employee 2035 (whose last name is Bond). Use the empty function for the CLOB, and provide NULL as the value for the BLOB.

```
INSERT INTO  personnel
VALUES (2034, 'Allen', empty_clob(), NULL);

INSERT INTO  personnel
VALUES (2035, 'Bond', empty_clob(), NULL);

1 row created.

1 row created.
```

3. Examine and execute the `E:\labs\PLPU\labs\lab_09_03.sql` script. The script creates a table named `REVIEW_TABLE`. This table contains annual review information for each employee. The script also contains two statements to insert review details for two employees.

```
CREATE TABLE review_table (
 employee_id number,
 ann_review  VARCHAR2(2000));

INSERT INTO review_table
VALUES (2034,
       'Very good performance this year. '||
       'Recommended to increase salary by $500');
INSERT INTO review_table
VALUES (2035,
       'Excellent performance this year. '||
       'Recommended to increase salary by $1000');

COMMIT;

Table created.

1 row created.

1 row created.

Commit complete.
```

4. Update the `PERSONNEL` table.

   a. Populate the `CLOB` for the first row by using the following subquery in an `UPDATE` statement:

```
SELECT ann_review
FROM review_table
WHERE  employee_id = 2034;
```

```
UPDATE personnel
 SET review = (SELECT ann_review
               FROM   review_table
               WHERE  employee_id = 2034)
 WHERE last_name = 'Allen';

1 row updated.
```

b. Populate the CLOB for the second row, using PL/SQL and the DBMS_LOB package. Use the following SELECT statement to provide a value for the LOB locator.

```
SELECT ann_review
FROM   review_table
WHERE  employee_id = 2035;
```

```
DECLARE
  lobloc CLOB;
  text VARCHAR2(2000);
  amount NUMBER ;
  offset INTEGER;
BEGIN
  SELECT ann_review INTO text
  FROM review_table
  WHERE employee_id = 2035;
  SELECT review INTO lobloc
  FROM personnel
  WHERE last_name = 'Bond' FOR UPDATE;
  offset := 1;
  amount := length(text);
  DBMS_LOB.WRITE (lobloc, amount, offset, text);
  COMMIT;
END;
/

PL/SQL procedure successfully completed.
```

If you have time, complete the following exercise:

5. Create a procedure that adds a locator to a binary file into the PICTURE column of the COUNTRIES table. The binary file is a picture of the country flag. The image files are named after the country IDs. You need to load an image file locator into all rows in the Europe region (REGION_ID = 1) in the COUNTRIES table. A DIRECTORY object called COUNTRY_PIC referencing the location of the binary files has to be created for you.

   a. Add the image column to the COUNTRIES table using:
   ALTER TABLE countries ADD (picture BFILE);

```
ALTER TABLE countries ADD (picture BFILE);

Table altered.
```

Alternatively, use the E:\labs\PLPU\labs\ Lab_09_05_a.sql file.

Practice 9: Solutions (continued)

b. Create a PL/SQL procedure called `load_country_image` that uses the `DBMS_LOB.FILEEXISTS` to test whether the country picture file exists. If the file exists, then set the `BFILE` locator for the file in the `PICTURE` column; otherwise, display a message that the file does not exist. Use the `DBMS_OUTPUT` package to report file size information for each image associated with the `PICTURE` column.

```
CREATE OR REPLACE PROCEDURE load_country_image (dir IN VARCHAR2) IS
  file           BFILE;
  filename       VARCHAR2(40);
  rec_number     NUMBER;
  file_exists    BOOLEAN;
  CURSOR country_csr IS
    SELECT country_id
    FROM countries
    WHERE region_id = 1
    FOR UPDATE;
BEGIN
  DBMS_OUTPUT.PUT_LINE('LOADING LOCATORS TO IMAGES...');
  FOR rec IN country_csr
  LOOP
    filename := rec.country_id || '.gif';
    file := BFILENAME(dir, filename);
    file_exists := (DBMS_LOB.FILEEXISTS(file) = 1);
    IF file_exists THEN
     DBMS_LOB.FILEOPEN(file);
     UPDATE countries
       SET picture = file
       WHERE CURRENT OF country_csr;
     DBMS_OUTPUT.PUT_LINE('Set Locator to file: '|| filename ||
       ' Size: ' || DBMS_LOB.GETLENGTH(file));
     DBMS_LOB.FILECLOSE(file);
     rec_number := country_csr%ROWCOUNT;
    ELSE
     DBMS_OUTPUT.PUT_LINE('File ' || filename ||' does not exist');
    END IF;
  END LOOP;
  DBMS_OUTPUT.PUT_LINE('TOTAL FILES UPDATED: ' || rec_number);
  EXCEPTION
    WHEN OTHERS THEN
      DBMS_LOB.FILECLOSE(file);
      DBMS_OUTPUT.PUT_LINE('Error: '|| to_char(SQLCODE) || SQLERRM);
END load_country_image;
/
SHOW ERRORS

Procedure created.

No errors.
```

Practice 9: Solutions (continued)

c. Invoke the procedure by passing the name of the directory object `COUNTRY_PIC` as a string literal parameter value.

```
SET SERVEROUTPUT ON
EXECUTE load_country_image('COUNTRY_PIC')

LOADING LOCATORS TO IMAGES...
Set Locator to file: BE.gif Size: 1397
Set Locator to file: CH.gif Size: 1202
Set Locator to file: DE.gif Size: 1271
Set Locator to file: DK.gif Size: 1327
Set Locator to file: FR.gif Size: 1337
Set Locator to file: IT.gif Size: 1322
Set Locator to file: NL.gif Size: 1205
Set Locator to file: UK.gif Size: 2489
TOTAL FILES UPDATED: 8
PL/SQL procedure successfully completed.
```

Practice 10: Solutions

1. The rows in the JOBS table store a minimum salary and a maximum salary allowed for different JOB_ID values. You are asked to write code to ensure that employees' salaries fall within the range allowed for their job type, for insert and update operations.

   a. Write a procedure called CHECK_SALARY that accepts two parameters, one for an employee's job ID string and the other for the salary. The procedure uses the job ID to determine the minimum and maximum salary for the specified job. If the salary parameter does not fall within the salary range of the job, inclusive of the minimum and maximum, then it should raise an application exception, with the message Invalid salary <sal>. Salaries for job <jobid> must be between <min> and <max>. Replace the various items in the message with values supplied by parameters and variables populated by queries. Save the file.

```
CREATE OR REPLACE PROCEDURE check_salary (the_job VARCHAR2, the_salary
NUMBER) IS
  minsal jobs.min_salary%type;
  maxsal jobs.max_salary%type;
BEGIN
  SELECT min_salary, max_salary INTO minsal, maxsal
  FROM jobs
  WHERE job_id = UPPER(the_job);
  IF the_salary NOT BETWEEN minsal AND maxsal THEN
    RAISE_APPLICATION_ERROR(-20100,
      'Invalid salary $'||the_salary||'. '||
      'Salaries for job '|| the_job ||
      ' must be between $'|| minsal ||' and $' || maxsal);
  END IF;
END;
/
SHOW ERRORS

Procedure created.

No errors.
```

   b. Create a trigger called CHECK_SALARY_TRG on the EMPLOYEES table that fires before an INSERT or UPDATE operation on each row. The trigger must call the CHECK_SALARY procedure to carry out the business logic. The trigger should pass the new job ID and salary to the procedure parameters.

```
CREATE OR REPLACE TRIGGER check_salary_trg
BEFORE INSERT OR UPDATE OF job_id, salary
ON employees
FOR EACH ROW
BEGIN
  check_salary(:new.job_id, :new.salary);
END;
/
SHOW ERRORS
```

```
Trigger created.

No errors.
```

2. Test the CHECK_SAL_TRG using the following cases:

   a. Using your EMP_PKG.ADD_EMPLOYEE procedure, add employee Eleanor Beh in department 30. What happens and why?

```
EXECUTE emp_pkg.add_employee('Eleanor', 'Beh', 30)

BEGIN emp_pkg.add_employee('Eleanor', 'Beh', 30); END;

*

ERROR at line 1:
ORA-20100: Invalid salary $1000. Salaries for job SA_REP must be between
$6000 and $12000
ORA-06512: at "ORA1.CHECK_SALARY", line 9
ORA-06512: at "ORA1.CHECK_SALARY_TRG", line 2
ORA-04088: error during execution of trigger 'ORA1.CHECK_SALARY_TRG'
ORA-06512: at "ORA1.EMP_PKG", line 33
ORA-06512: at "ORA1.EMP_PKG", line 50
ORA-06512: at line 1
```

   The trigger raises an exception because the `EMP_PKG.ADD_EMPLOYEE` procedure invokes an overloaded version of itself that uses the default salary of $1,000 and the default job ID of `SA_REP`. However, the `JOBS` table stores a minimum salary of $6,000 for the `SA_REP` job type.

   b. Update the salary of employee 115 to $2,000. In a separate update operation, change the employee job ID to HR_REP. What happens in each case?

```
UPDATE employees
  SET salary = 2000
WHERE employee_id = 115;

UPDATE employees
       *

ERROR at line 1:
ORA-20100: Invalid salary $2000. Salaries for job PU_CLERK must be
between $2500 and $5500
ORA-06512: at "ORA1.CHECK_SALARY", line 9
ORA-06512: at "ORA1.CHECK_SALARY_TRG", line 2
ORA-04088: error during execution of trigger 'ORA1.CHECK_SALARY_TRG'
```

Practice 10: Solutions (continued)

```
UPDATE employees
  SET job_id = 'HR_REP'
WHERE employee_id = 115;

UPDATE employees
        *

ERROR at line 1:
ORA-20100: Invalid salary $3100. Salaries for job HR_REP must be between
$4000 and $9000
ORA-06512: at "ORA1.CHECK_SALARY", line 9
ORA-06512: at "ORA1.CHECK_SALARY_TRG", line 2
ORA-04088: error during execution of trigger 'ORA1.CHECK_SALARY_TRG'
```

The first update statement fails to set the salary to $2,000. The check salary trigger rule fails the update operation because the new salary for employee 115 is less than the minimum allowed for the `PU_CLERK` job.

The second update fails to change the employee's job because the current employee's salary of $3,100 is less than the minimum for the new `HR_REP` job.

c. Update the salary of employee 115 to $2,800. What happens?

```
UPDATE employees
  SET salary = 2800
WHERE employee_id = 115;

1 row updated.
```

The update operation is successful because the new salary falls within the acceptable range for the current job ID.

3. Update the `CHECK_SALARY_TRG` trigger to fire only when the job ID or salary values have actually changed.

   a. Implement the business rule using a `WHEN` clause to check whether the `JOB_ID` or `SALARY` values have changed.
   Note: Make sure that the condition handles the `NULL` in the `OLD.column_name` values if an `INSERT` operation is performed; otherwise, an insert operation will fail.

```
CREATE OR REPLACE TRIGGER check_salary_trg
BEFORE INSERT OR UPDATE OF job_id, salary
ON employees FOR EACH ROW
WHEN (new.job_id <> NVL(old.job_id,'?') OR
      new.salary <> NVL(old.salary,0))
BEGIN
  check_salary(:new.job_id, :new.salary);
END;
/
```

Practice 10: Solutions (continued)

```
SHOW ERRORS

Trigger created.

No errors.
```

b. Test the trigger by executing `EMP_PKG.ADD_EMPLOYEE` procedure with the following parameter values: `first_name='Eleanor', last name='Beh', email='EBEH', job='IT_PROG', sal=5000`.

```
BEGIN
  emp_pkg.add_employee('Eleanor', 'Beh', 'EBEH',
                       job => 'IT_PROG', sal => 5000);
END;
/

PL/SQL procedure successfully completed.
```

c. Update employees with the `IT_PROG` job by incrementing their salary by $2,000. What happens?

```
UPDATE employees
  SET salary = salary + 2000
WHERE job_id = 'IT_PROG';

UPDATE employees
         *

ERROR at line 1:
ORA-20100: Invalid salary $11000. Salaries for job IT_PROG must be
between $4000 and $10000
ORA-06512: at "ORA1.CHECK_SALARY", line 9
ORA-06512: at "ORA1.CHECK_SALARY_TRG", line 2
ORA-04088: error during execution of trigger 'ORA1.CHECK_SALARY_TRG'
```

An employee's salary in the specified job type exceeds the maximum salary for that job type. No employee salaries in the `IT_PROG` job type are updated.

d. Update the salary to $9,000 for `Eleanor Beh`.
Hint: Use an `UPDATE` statement with a subquery in the `WHERE` clause. What happens?

```
UPDATE employees
  SET salary = 9000
WHERE employee_id = (SELECT employee_id
                     FROM employees
                     WHERE last_name = 'Beh');

1 row updated
```

The update operation is successful because the salary is valid for the employee's job type.

e. Change the job of `Eleanor Beh` to `ST_MAN` using another `UPDATE` statement with a subquery. What happens?

```
UPDATE employees
  set job_id = 'ST_MAN'
WHERE employee_id = (SELECT employee_id
                     FROM employees
                     WHERE last_name = 'Beh');

UPDATE employees
         *

ERROR at line 1:
ORA-20100: Invalid salary $9000. Salaries for job ST_MAN must be between
$5500 and $8500
ORA-06512: at "ORA1.CHECK_SALARY", line 9
ORA-06512: at "ORA1.CHECK_SALARY_TRG", line 2
ORA-04088: error during execution of trigger 'ORA1.CHECK_SALARY_TRG'
```

The maximum salary of the new job type is less than the employee's current salary. Therefore, the operation update fails.

Practice 10: Solutions (continued)

4. You are asked to prevent employees from being deleted during business hours.

   a. Write a statement trigger called `DELETE_EMP_TRG` on the `EMPLOYEES` table to prevent rows from being deleted during weekday business hours, which are from 9:00 a.m. to 6:00 p.m.

```
CREATE OR REPLACE TRIGGER delete_emp_trg
BEFORE DELETE ON employees
DECLARE
  the_day VARCHAR2(3) := TO_CHAR(SYSDATE, 'DY');
  the_hour PLS_INTEGER := TO_NUMBER(TO_CHAR(SYSDATE, 'HH24'));
BEGIN
   IF (the_hour BETWEEN 9 AND 18) AND (the_day NOT IN ('SAT','SUN')) THEN
     RAISE_APPLICATION_ERROR(-20150,
      'Employee records cannot be deleted during the week 9am and 6pm');
   END IF;
END;
/
SHOW ERRORS

Trigger created.

No errors.
```

   b. Attempt to delete employees with `JOB_ID` of `SA_REP` who are not assigned to a department.
   Note: This is employee `Grant` with ID 178.

```
DELETE FROM employees
  WHERE job_id = 'SA_REP'
  AND   department_id IS NULL;

DELETE FROM employees
         *

ERROR at line 1:
ORA-20150: Employee records cannot be deleted during the week 9am and 6pm
ORA-06512: at "ORA1.DELETE_EMP_TRG", line 6
ORA-04088: error during execution of trigger 'ORA1.DELETE_EMP_TRG'
```

Practice 11: Solutions

1. Employees receive an automatic increase in salary if the minimum salary for a job is increased to a value larger than their current salary. Implement this requirement through a package procedure called by a trigger on the JOBS table. When you attempt to update the minimum salary in the JOBS table and try to update the employee's salary, the CHECK_SALARY trigger attempts to read the JOBS table, which is subject to change, and you get a mutating table exception that is resolved by creating a new package and additional triggers.

   a. Update your EMP_PKG package (from Practice 7) by adding a procedure called SET_SALARY that updates the employees' salaries. The procedure accepts two parameters: the job ID for those salaries that may have to be updated, and the new minimum salary for the job ID. The procedure sets all the employee salaries to the minimum for their job if their current salary is less than the new minimum value.

```
CREATE OR REPLACE PACKAGE emp_pkg IS
  TYPE emp_tabtype IS TABLE OF employees%ROWTYPE;
  PROCEDURE add_employee(
    first_name employees.first_name%TYPE,
    last_name employees.last_name%TYPE,
    email employees.email%TYPE,
    job employees.job_id%TYPE DEFAULT 'SA_REP',
    mgr employees.manager_id%TYPE DEFAULT 145,
    sal employees.salary%TYPE DEFAULT 1000,
    comm employees.commission_pct%TYPE DEFAULT 0,
    deptid employees.department_id%TYPE DEFAULT 30);
  PROCEDURE add_employee(
    first_name employees.first_name%TYPE,
    last_name employees.last_name%TYPE,
    deptid employees.department_id%TYPE);
  PROCEDURE get_employee(
    empid IN employees.employee_id%TYPE,
    sal OUT employees.salary%TYPE,
    job OUT employees.job_id%TYPE);
  FUNCTION get_employee(emp_id employees.employee_id%type)
    return employees%rowtype;
  FUNCTION get_employee(family_name employees.last_name%type)
    return employees%rowtype;
  PROCEDURE get_employees(dept_id employees.department_id%type);
  PROCEDURE init_departments;
  PROCEDURE print_employee(emprec employees%rowtype);
  PROCEDURE set_salary(jobid VARCHAR2, min_salary NUMBER);
END emp_pkg;
/
SHOW ERRORS
```

Practice 11: Solutions (continued)

```
CREATE OR REPLACE PACKAGE BODY emp_pkg IS
  TYPE boolean_tabtype IS TABLE OF BOOLEAN
    INDEX BY BINARY_INTEGER;
  valid_departments boolean_tabtype;
  emp_table         emp_tabtype;

  FUNCTION valid_deptid(deptid IN departments.department_id%TYPE)
   RETURN BOOLEAN;
  PROCEDURE add_employee(
    first_name employees.first_name%TYPE,
    last_name employees.last_name%TYPE,
    email employees.email%TYPE,
    job employees.job_id%TYPE DEFAULT 'SA_REP',
    mgr employees.manager_id%TYPE DEFAULT 145,
    sal employees.salary%TYPE DEFAULT 1000,
    comm employees.commission_pct%TYPE DEFAULT 0,
    deptid employees.department_id%TYPE DEFAULT 30) IS

    PROCEDURE audit_newemp IS
      PRAGMA AUTONOMOUS_TRANSACTION;
      user_id VARCHAR2(30) := USER;
    BEGIN
      INSERT INTO log_newemp (entry_id, user_id, log_time, name)
      VALUES (log_newemp_seq.NEXTVAL, user_id, sysdate,
              first_name||' '||last_name);
      COMMIT;
    END audit_newemp;

  BEGIN
    IF valid_deptid(deptid) THEN
      audit_newemp;
      INSERT INTO employees(employee_id, first_name, last_name, email,
        job_id,manager_id,hire_date,salary,commission_pct,department_id)
      VALUES (employees_seq.NEXTVAL, first_name, last_name, email,
        job, mgr, TRUNC(SYSDATE), sal, comm, deptid);
    ELSE
      RAISE_APPLICATION_ERROR (-20204,
                              'Invalid department ID. Try again.');
    END IF;
  END add_employee;

  PROCEDURE add_employee(
    first_name employees.first_name%TYPE,
    last_name employees.last_name%TYPE,
    deptid employees.department_id%TYPE) IS
    email employees.email%type;
  BEGIN
    email := UPPER(SUBSTR(first_name, 1, 1)||SUBSTR(last_name, 1, 7));
    add_employee(first_name, last_name, email, deptid => deptid);
  END;
```

Practice 11: Solutions (continued)

```
PROCEDURE get_employee(
  empid IN employees.employee_id%TYPE,
  sal OUT employees.salary%TYPE,
  job OUT employees.job_id%TYPE) IS
BEGIN
  SELECT salary, job_id
  INTO sal, job
  FROM employees
  WHERE employee_id = empid;
END get_employee;

FUNCTION get_employee(emp_id employees.employee_id%type)
  return employees%rowtype IS
  emprec employees%rowtype;
BEGIN
  SELECT * INTO emprec
  FROM employees
  WHERE employee_id = emp_id;
  RETURN emprec;
END;

FUNCTION get_employee(family_name employees.last_name%type)
  return employees%rowtype IS
  emprec employees%rowtype;
BEGIN
  SELECT * INTO emprec
  FROM employees
  WHERE last_name = family_name;
  RETURN emprec;
END;

PROCEDURE get_employees(dept_id employees.department_id%type) IS
BEGIN
  SELECT * BULK COLLECT INTO emp_table
  FROM EMPLOYEES
  WHERE department_id = dept_id;
END;

PROCEDURE init_departments IS
BEGIN
  FOR rec IN (SELECT department_id FROM departments)
  LOOP
    valid_departments(rec.department_id) := TRUE;
  END LOOP;
END;
```

Practice 11: Solutions (continued)

```
  PROCEDURE print_employee(emprec employees%rowtype) IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE(emprec.department_id ||' '||
                         emprec.employee_id||' '||
                         emprec.first_name||' '||
                         emprec.last_name||' '||
                         emprec.job_id||' '||
                         emprec.salary);
  END;

  PROCEDURE show_employees IS
  BEGIN
    IF emp_table IS NOT NULL THEN
      DBMS_OUTPUT.PUT_LINE('Employees in Package table');
      FOR i IN 1 .. emp_table.COUNT
      LOOP
        print_employee(emp_table(i));
      END LOOP;
    END IF;
  END show_employees;

  FUNCTION valid_deptid(deptid IN departments.department_id%TYPE)
   RETURN BOOLEAN IS
    dummy PLS_INTEGER;
  BEGIN
    RETURN valid_departments.exists(deptid);
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
    RETURN FALSE;
  END valid_deptid;

  PROCEDURE set_salary(jobid VARCHAR2, min_salary NUMBER) IS
    CURSOR empcsr IS
      SELECT employee_id
      FROM employees
      WHERE job_id = jobid AND salary < min_salary;
  BEGIN
    FOR emprec IN empcsr
    LOOP
      UPDATE employees
        SET salary = min_salary
      WHERE employee_id = emprec.employee_id;
    END LOOP;
  END set_salary;

BEGIN
  init_departments;
END emp_pkg;
/
SHOW ERRORS
```

Practice 11: Solutions (continued)

```
Package created.

No errors.

Package body created.

No errors.
```

b. Create a row trigger named UPD_MINSALARY_TRG on the JOBS table that invokes the EMP_PKG.SET_SALARY procedure, when the minimum salary in the JOBS table is updated for a specified job ID.

```
CREATE OR REPLACE TRIGGER upd_minsalary_trg
AFTER UPDATE OF min_salary ON JOBS
FOR EACH ROW
BEGIN
  emp_pkg.set_salary(:new.job_id, :new.min_salary);
END;
/
SHOW ERRORS

Trigger created.

No errors.
```

c. Write a query to display the employee ID, last name, job ID, current salary, and minimum salary for employees who are programmers—that is, their JOB_ID is 'IT_PROG'. Then update the minimum salary in the JOBS table to increase it by $1,000. What happens?

```
SELECT employee_id, last_name, salary
FROM employees
WHERE job_id = 'IT_PROG';

UPDATE jobs
 SET min_salary = min_salary + 1000
WHERE job_id = 'IT_PROG';
```

EMPLOYEE_ID	LAST_NAME	SALARY
103	Hunold	9000
104	Ernst	6000
105	Austin	4800
106	Pataballa	4800
107	Lorentz	4200
226	Beh	9000

```
6 rows selected.
```

Practice 11: Solutions (continued)

```
UPDATE jobs
       *

ERROR at line 1:
ORA-04091: table ORA1.JOBS is mutating, trigger/function may not see it
ORA-06512: at "ORA1.CHECK_SALARY", line 5
ORA-06512: at "ORA1.CHECK_SALARY_TRG", line 2
ORA-04088: error during execution of trigger 'ORA1.CHECK_SALARY_TRG'
ORA-06512: at "ORA1.EMP_PKG", line 140
ORA-06512: at "ORA1.UPD_MINSALARY_TRG", line 2
ORA-04088: error during execution of trigger 'ORA1.UPD_MINSALARY_TRG'
```

The update of the `MIN_SALARY` column for job `'IT_PROG'` fails because the `UPD_MINSALARY_TRG` trigger on the `JOBS` table attempts to update the employees' salaries by calling the `EMP_PKG.SET_SALARY` procedure. The `SET_SALARY` procedure causes the `CHECK_SALARY_TRG` trigger to fire (a cascading effect). `CHECK_SALARY_TRG` calls the `CHECK_SALARY` procedure, which attempts to read the `JOBS` table data, thus encountering the mutating table exception on the `JOBS` table, which is the table that is subject to the original `UPDATE` operation.

2. To resolve the mutating table issue, you create `JOBS_PKG` to maintain in memory a copy of the rows in the `JOBS` table. Then the `CHECK_SALARY` procedure is modified to use the package data rather than issue a query on a table that is mutating to avoid the exception. However, a `BEFORE INSERT OR UPDATE` statement trigger must be created on the `EMPLOYEES` table to initialize the `JOBS_PKG` package state before the `CHECK_SALARY` row trigger is fired.

   a. Create a new package called `JOBS_PKG` with the following specification.

```
PROCEDURE initialize;
FUNCTION get_minsalary(jobid VARCHAR2) RETURN NUMBER;
FUNCTION get_maxsalary(jobid VARCHAR2) RETURN NUMBER;
PROCEDURE set_minsalary(jobid VARCHAR2,min_salary NUMBER);
PROCEDURE set_maxsalary(jobid VARCHAR2,max_salary NUMBER);
```

```
CREATE OR REPLACE PACKAGE jobs_pkg IS
  PROCEDURE initialize;
  FUNCTION get_minsalary(jobid VARCHAR2) RETURN NUMBER;
  FUNCTION get_maxsalary(jobid VARCHAR2) RETURN NUMBER;
  PROCEDURE set_minsalary(jobid VARCHAR2, min_salary NUMBER);
  PROCEDURE set_maxsalary(jobid VARCHAR2, max_salary NUMBER);
END jobs_pkg;
/
SHOW ERRORS

Package created.

No errors.
```

Practice 11: Solutions (continued)

b. Implement the body of the JOBS_PKG where:
You declare a private PL/SQL index-by table called jobs_tabtype that is indexed by a string type based on JOBS.JOB_ID%TYPE.
You declare a private variable called jobstab based on jobs_tabtype.

The INITIALIZE procedure reads the rows in the JOBS table by using a cursor loop, and uses the JOB_ID value for the jobstab index that is assigned its corresponding row.
The GET_MINSALARY function uses a jobid parameter as an index to the jobstab and returns the min_salary for that element.
The GET_MAXSALARY function uses a jobid parameter as an index to the jobstab and returns the max_salary for that element.
The SET_MINSALARY procedure uses its jobid as an index to the jobstab to set the min_salary field of its element to the value in the min_salary parameter.
The SET_MAXSALARY procedure uses its jobid as an index to the jobstab to set the max_salary field of its element to the value in the max_salary parameter.

```
CREATE OR REPLACE PACKAGE BODY jobs_pkg IS
  TYPE jobs_tabtype IS TABLE OF jobs%rowtype
    INDEX BY jobs.job_id%type;
  jobstab jobs_tabtype;

  PROCEDURE initialize IS
  BEGIN
    FOR jobrec IN (SELECT * FROM jobs)
    LOOP
      jobstab(jobrec.job_id) := jobrec;
    END LOOP;
  END initialize;

  FUNCTION get_minsalary(jobid VARCHAR2) RETURN NUMBER IS
  BEGIN
    RETURN jobstab(jobid).min_salary;
  END get_minsalary;

  FUNCTION get_maxsalary(jobid VARCHAR2) RETURN NUMBER IS
  BEGIN
    RETURN jobstab(jobid).max_salary;
  END get_maxsalary;

  PROCEDURE set_minsalary(jobid VARCHAR2, min_salary NUMBER) IS
  BEGIN
    jobstab(jobid).max_salary := min_salary;
  END set_minsalary;
```

```
  PROCEDURE set_maxsalary(jobid VARCHAR2, max_salary NUMBER) IS
  BEGIN
    jobstab(jobid).max_salary := max_salary;
  END set_maxsalary;

END jobs_pkg;
/
SHOW ERRORS

Package body created.

No errors.
```

c. Copy the CHECK_SALARY procedure from Practice 10, Exercise 1a, and modify the code by replacing the query on the JOBS table with statements to set the local minsal and maxsal variables with values from the JOBS_PKG data by calling the appropriate GET_*SALARY functions. This step should eliminate the mutating trigger exception.

```
CREATE OR REPLACE PROCEDURE check_salary (the_job VARCHAR2, the_salary
NUMBER) IS
  minsal jobs.min_salary%type;
  maxsal jobs.max_salary%type;
BEGIN
  /*
  ** Commented out to avoid mutating trigger exception on the JOBS table
  SELECT min_salary, max_salary INTO minsal, maxsal
  FROM jobs
  WHERE job_id = UPPER(the_job);
  */
  minsal := jobs_pkg.get_minsalary(UPPER(the_job));
  maxsal := jobs_pkg.get_maxsalary(UPPER(the_job));
  IF the_salary NOT BETWEEN minsal AND maxsal THEN
    RAISE_APPLICATION_ERROR(-20100,
      'Invalid salary $'||the_salary||'. '||
      'Salaries for job '|| the_job ||
      ' must be between $'|| minsal ||' and $' || maxsal);
  END IF;
END;
/
SHOW ERRORS

Procedure created.

No errors.
```

d. Implement a BEFORE INSERT OR UPDATE statement trigger called INIT_JOBPKG_TRG that uses the CALL syntax to invoke the JOBS_PKG.INITIALIZE procedure to ensure that the package state is current before the DML operations are performed.

```
CREATE OR REPLACE TRIGGER init_jobpkg_trg
BEFORE INSERT OR UPDATE ON jobs
CALL jobs_pkg.initialize
/
SHOW ERRORS

Trigger created.

No errors.
```

e. Test the code changes by executing the query to display the employees who are programmers, and then issue an update statement to increase the minimum salary of the IT_PROG job type by 1000 in the JOBS table, followed by a query on the employees with the IT_PROG job type to check the resulting changes. Which employees' salaries have been set to the minimum for their job?

```
SELECT employee_id, last_name, salary
FROM employees
WHERE job_id = 'IT_PROG';

UPDATE jobs
  SET min_salary = min_salary + 1000
WHERE job_id = 'IT_PROG';

SELECT employee_id, last_name, salary
FROM employees
WHERE job_id = 'IT_PROG';
```

EMPLOYEE_ID	LAST_NAME	SALARY
103	Hunold	9000
104	Ernst	6000
105	Austin	4800
106	Pataballa	4800
107	Lorentz	4200
226	Beh	9000

```
6 rows selected.

1 row updated.
```

EMPLOYEE_ID	LAST_NAME	SALARY
103	Hunold	9000
104	Ernst	6000
105	Austin	5000
106	Pataballa	5000
107	Lorentz	5000
226	Beh	9000

```
6 rows selected.
```

The employees with last names `Austin`, `Pataballa`, and `Lorentz` have all had their salaries updated. No exception ocurred during this process, and you implemented a solution for the mutating table trigger exception.

3. Because the `CHECK_SALARY` procedure is fired by `CHECK_SALARY_TRG` before inserting or updating an employee, you must check whether this still works as expected.

   a. Test this by adding a new employee using `EMP_PKG.ADD_EMPLOYEE` with the following parameters: (‘`Steve`’, ‘`Morse`’, ‘`SMORSE`’, `sal => 6500`). What happens?

```
EXECUTE emp_pkg.add_employee('Steve', 'Morse', 'SMORSE', sal => 6500)

BEGIN emp_pkg.add_employee('Steve', 'Morse', 'SMORSE', sal => 6500); END;

*

ERROR at line 1:
ORA-01403: no data found
ORA-01403: no data found
ORA-06512: at "ORA1.JOBS_PKG", line 16
ORA-06512: at "ORA1.CHECK_SALARY", line 11
ORA-06512: at "ORA1.CHECK_SALARY_TRG", line 2
ORA-04088: error during execution of trigger 'ORA1.CHECK_SALARY_TRG'
ORA-06512: at "ORA1.EMP_PKG", line 33
ORA-06512: at line 1
```

The problem here is that the `CHECK_SALARY` procedure attempts to read the value of package state variables that have not yet been initialized. This is because it had been modified to read the miniumum and maximum salaries from `JOBS_PK`, which should store the data in a PL/SQL table. When `CHECK_SALARY` attempts to call `JOBS_PKG.GET_MINSALARY` and `JOBS_PKG.GET_MAXSALARY`, these return `NO_DATA_FOUND` exceptions that cause the trigger and the insert operation to fail. This can be resolved with a `BEFORE` statement trigger that calls `JOBS_PKG.INITIALIZE` to ensure that the `JOBS_PKG` state is set before you read it. This is done in the next exercise (3b).

b. To correct the problem encountered when adding or updating an employee, create a `BEFORE INSERT OR UPDATE` statement trigger called `EMPLOYEE_INITJOBS_TRG` on the `EMPLOYEES` table that calls the `JOBS_PKG.INITIALIZE` procedure. Use the `CALL` syntax in the trigger body.

```
CREATE TRIGGER employee_initjobs_trg
BEFORE INSERT OR UPDATE OF job_id, salary ON employees
CALL jobs_pkg.initialize
/

Trigger created.
```

c. Test the trigger by adding employee Steve Morse again. Confirm the inserted record in the `employees` table by displaying the employee ID, first and last names, salary, job ID, and department ID.

```
EXECUTE emp_pkg.add_employee('Steve', 'Morse', 'SMORSE', sal => 6500)

PL/SQL procedure successfully completed.

SELECT employee_id, first_name, last_name, salary, job_id, department_id
FROM employees
WHERE last_name = 'Morse';
```

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	SALARY	JOB_ID	DEPARTMENT_ID
241	Steve	Morse	6500	SA_REP	30

Practice 12: Solutions

1. Alter the PLSQL_COMPILER_FLAGS parameter to enable native compilation for your session, and compile any subprogram that you have written.

   a. Execute the ALTER SESSION command to enable native compilation.

```
ALTER SESSION SET PLSQL_COMPILER_FLAGS = 'NATIVE';

Session altered.
```

   b. Compile the EMPLOYEE_REPORT procedure. What occurs during compilation?

```
ALTER PROCEDURE employee_report COMPILE;

Procedure altered.
```

   A shared library is generated in a directory specified by the database parameter, `plsql_native_library_dir`. The library name is prefixed with the object name and user compiling it, as in the following: `EMPLOYEE_REPORT__ORA1__P__50344.so`.

   c. Execute the EMPLOYEE_REPORT with the value 'UTL_FILE' as the first parameter, and 'native_salrepXX.txt' where XX is your student number.

```
EXECUTE employee_report('UTL_FILE', 'native_salrep01.txt')

PL/SQL procedure successfully completed.
```

   d. Switch compilation to use interpreted compilation

```
ALTER SESSION SET PLSQL_COMPILER_FLAGS = 'INTERPRETED';

Session altered.
```

2. In COMPILE_PKG (from Practice 6), add an overloaded version of the procedure called MAKE, which will compile a named procedure, function, or package.

   a. In the specification, declare a MAKE procedure that accepts two string arguments, one for the name of the PL/SQL construct and the other for the type of PL/SQL program, such as PROCEDURE, FUNCTION, PACKAGE, or PACKAGE BODY.

```
CREATE OR REPLACE PACKAGE compile_pkg IS
  dir VARCHAR2(100) := 'UTL_FILE';
  PROCEDURE make(name VARCHAR2);
  PROCEDURE make(name VARCHAR2, objtype VARCHAR2);
  PROCEDURE regenerate(name VARCHAR2);
END compile_pkg;
/
SHOW ERRORS

Package created.

No errors.
```

b. In the body, write the MAKE procedure to call the DBMS_WARNINGS package to suppress the PERFORMANCE category. However, save the current compiler warning settings before you alter them. Then write an EXECUTE IMMEDIATE statement to compile the PL/SQL object using an appropriate ALTER...COMPILE statement with the supplied parameter values. Finally, restore the compiler warning settings that were in place for the calling environment before the procedure is invoked.

```
CREATE OR REPLACE PACKAGE BODY compile_pkg IS

  PROCEDURE execute(stmt VARCHAR2) IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE(stmt);
    EXECUTE IMMEDIATE stmt;
  END;

  FUNCTION get_type(name VARCHAR2) RETURN VARCHAR2 IS
    proc_type VARCHAR2(30) := NULL;
  BEGIN
    /*
     * The ROWNUM = 1 is added to the condition
     * to ensure only one row is returned if the
     * name represents a PACKAGE, which may also
     * have a PACKAGE BODY. In this case, we can
     * only compile the complete package, but not
     * the specification or body as separate
     * components.
     */
    SELECT object_type INTO proc_type
    FROM user_objects
    WHERE object_name = UPPER(name)
    AND ROWNUM = 1;
    RETURN proc_type;
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
      RETURN NULL;
  END;

  PROCEDURE make(name VARCHAR2) IS
    stmt       VARCHAR2(100);
    proc_type  VARCHAR2(30) := get_type(name);
  BEGIN
    IF proc_type IS NOT NULL THEN
      stmt := 'ALTER '|| proc_type ||' '|| name ||' COMPILE';
      execute(stmt);
    ELSE
      RAISE_APPLICATION_ERROR(-20001,
         'Subprogram '''|| name ||''' does not exist');
    END IF;
  END make;
```

Practice 12: Solutions (continued)

```
  PROCEDURE make(name VARCHAR2, objtype VARCHAR2) IS
    stmt       VARCHAR2(100);
    warn_value varchar2(200);
  BEGIN
      stmt := 'ALTER '|| objtype ||' '|| name ||' COMPILE';
      warn_value := dbms_warning.get_warning_setting_string;
      dbms_warning.add_warning_setting_cat(
         'PERFORMANCE', 'DISABLE', 'SESSION');
      execute(stmt);
      dbms_warning.set_warning_setting_string(
         warn_value, 'SESSION');
  END make;

  PROCEDURE regenerate (name VARCHAR2) IS
    file UTL_FILE.FILE_TYPE;
    filename VARCHAR2(100) := LOWER(USER ||'_'|| name ||'.sql');
    proc_type  VARCHAR2(30) := get_type(name);
  BEGIN
    IF proc_type IS NOT NULL THEN
      file := UTL_FILE.FOPEN(dir, filename, 'w');
      UTL_FILE.PUT(file,
        DBMS_METADATA.GET_DDL(proc_type, UPPER(name)));
      UTL_FILE.FCLOSE(file);
    ELSE
      RAISE_APPLICATION_ERROR(-20001,
         'Object with '''|| name ||''' does not exist');
    END IF;

  END regenerate;

END compile_pkg;
/
SHOW ERRORS

Package body created.

No errors.
```

Practice 12: Solutions (continued)

3. Write a new PL/SQL package called TEST_PKG containing a procedure called GET_EMPLOYEES that uses an IN OUT argument.

   a. In the specification, declare the GET_EMPLOYEES procedure with two parameters, one input parameter specifying a department ID, and an IN OUT parameter specifying a PL/SQL table of employee rows.
   Hint: You have to declare a TYPE in the package specification for the PL/SQL table parameter's data type.

```
CREATE OR REPLACE PACKAGE test_pkg IS
  TYPE emp_tabtype IS TABLE OF employees%ROWTYPE;
  PROCEDURE get_employees(dept_id NUMBER, emps IN OUT emp_tabtype);
END test_pkg;
/
SHOW ERRORS

Package created.

No errors.
```

   b. In the package body, implement the GET_EMPLOYEES procedure to retrieve all the employee rows for a specified department into the PL/SQL table IN OUT parameter.
   Hint: Use the SELECT … BULK COLLECT INTO syntax to simplify the code.

```
CREATE OR REPLACE PACKAGE BODY test_pkg IS
  PROCEDURE get_employees(dept_id NUMBER, emps IN OUT emp_tabtype) IS
  BEGIN
     SELECT * BULK COLLECT INTO emps
     FROM employees
     WHERE department_id = dept_id;
  END get_employees;
END test_pkg;
/
SHOW ERRORS

Package body created.

No errors.
```

4. Use the ALTER SESSION statement to set the PLSQL_WARNINGS so that all compiler warning categories are enabled.

```
ALTER SESSION SET PLSQL_WARNINGS = 'ENABLE:ALL';

Session altered.
```

Practice 12: Solutions (continued)

5. Recompile the TEST_PKG created in an earlier task. What compiler warnings are displayed, if any?

```
ALTER PACKAGE test_pkg COMPILE;
SHOW ERRORS

SP2-0809: Package altered with compilation warnings
Errors for PACKAGE TEST_PKG:
```

LINE/COL	ERROR
3/43	PLW-07203: parameter 'EMPS' may benefit from use of the NOCOPY co mpiler hint

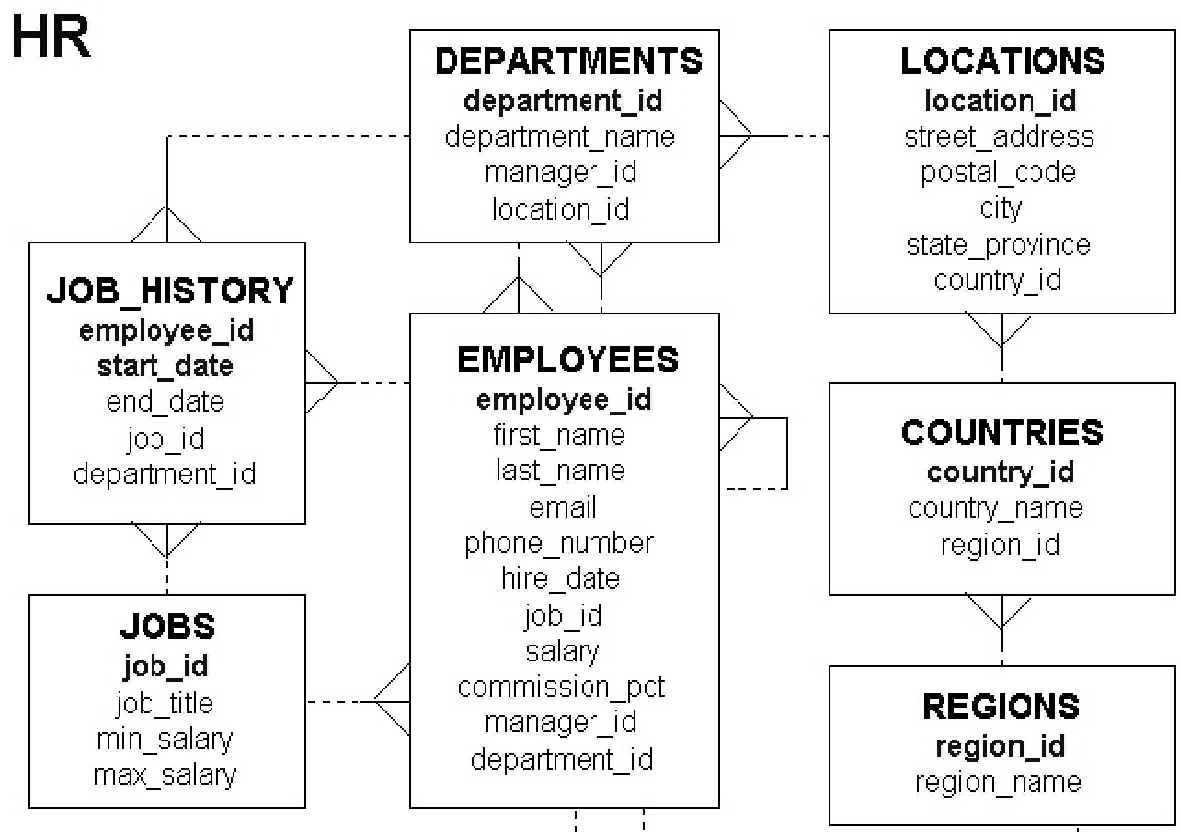
6. Write a PL/SQL anonymous block to compile the TEST_PKG package by using the overloaded COMPILE_PKG.MAKE procedure with two parameters. The anonymous block should display the current session warning string value before and after it invokes the COMPILE_PKG.MAKE procedure. Do you see any warning messages? Confirm your observations by executing the SHOW ERRORS PACKAGE command for the TEST_PKG.

```
BEGIN
  dbms_output.put_line('Warning level before compilation: '||
            dbms_warning.get_warning_setting_string);
  compile_pkg.make('TEST_PKG', 'PACKAGE');
  dbms_output.put_line('Warning level after compilation: '||
            dbms_warning.get_warning_setting_string);
END;
/
SHOW ERRORS PACKAGE test_pkg;

Warning level before compilation: ENABLE:ALL
ALTER PACKAGE TEST_PKG COMPILE
Warning level after compilation: ENABLE:ALL
PL/SQL procedure successfully completed.

No errors.
```

Note: The current warning level setting should be the same before and after the call to the COMPILE_PKG.MAKE procedure, which alters the settings to suppress warnings and restores the original setting before returning to the caller.

Table Descriptions and Data

Entity Relationship Diagram



Tables in the Schema

```
SELECT * FROM tab;
```

TNAME	TABTYPE	CLUSTERID
COUNTRIES	TABLE	
DEPARTMENTS	TABLE	
EMPLOYEES	TABLE	
EMP_DETAILS_VIEW	VIEW	
JOBS	TABLE	
JOB_HISTORY	TABLE	
LOCATIONS	TABLE	
REGIONS	TABLE	

8 rows selected.

REGIONS Table

```
DESCRIBE regions
```

Name	Null?	Type
REGION_ID	NOT NULL	NUMBER
REGION_NAME		VARCHAR2(25)

```
SELECT * FROM regions;
```

REGION_ID	REGION_NAME
1	Europe
2	Americas
3	Asia
4	Middle East and Africa

`COUNTRIES` Table

```
DESCRIBE countries
```

Name	Null?	Type
COUNTRY_ID	NOT NULL	CHAR(2)
COUNTRY_NAME		VARCHAR2(40)
REGION_ID		NUMBER

```
SELECT * FROM countries;
```

CO	COUNTRY_NAME	REGION_ID
AR	Argentina	2
AU	Australia	3
BE	Belgium	1
BR	Brazil	2
CA	Canada	2
CH	Switzerland	1
CN	China	3
DE	Germany	1
DK	Denmark	1
EG	Egypt	4
FR	France	1
HK	HongKong	3
IL	Israel	4
IN	India	3
CO	COUNTRY_NAME	REGION_ID
IT	Italy	1
JP	Japan	3
KW	Kuwait	4
MX	Mexico	2
NG	Nigeria	4
NL	Netherlands	1
SG	Singapore	3
UK	United Kingdom	1
US	United States of America	2
ZM	Zambia	4
ZW	Zimbabwe	4

25 rows selected.

`LOCATIONS` Table

```
DESCRIBE locations;
```

Name	Null?	Type
LOCATION_ID	NOT NULL	NUMBER(4)
STREET_ADDRESS		VARCHAR2(40)
POSTAL_CODE		VARCHAR2(12)
CITY	NOT NULL	VARCHAR2(30)
STATE_PROVINCE		VARCHAR2(25)
COUNTRY_ID		CHAR(2)

```
SELECT * FROM locations;
```

LOCATION_ID	STREET_ADDRESS	POSTAL_CODE	CITY	STATE_PROVINCE	CO
1000	1297 Via Cola di Rie	00989	Roma		IT
1100	93091 Calle della Testa	10934	Venice		IT
1200	2017 Shinjuku-ku	1689	Tokyo	Tokyo Prefecture	JP
1300	9450 Kamiya-cho	6823	Hiroshima		JP
1400	2014 Jabberwocky Rd	26192	Southlake	Texas	US
1500	2011 Interiors Blvd	99236	South San Francisco	California	US
1600	2007 Zagora St	50090	South Brunswick	New Jersey	US
1700	2004 Charade Rd	98199	Seattle	Washington	US
1800	147 Spadina Ave	M5V 2L7	Toronto	Ontario	CA
1900	6092 Boxwood St	YSW 9T2	Whitehorse	Yukon	CA
2000	40-5-12 Laogianggen	190518	Beijing		CN
2100	1298 Vileparle (E)	490231	Bombay	Maharashtra	IN
LOCATION_ID	STREET_ADDRESS	POSTAL_CODE	CITY	STATE_PROVINCE	CO
2400	8204 Arthur St		London		UK
2500	Magdalen Centre, The Oxford Science Park	OX9 9ZB	Oxford	Oxford	UK
2600	9702 Chester Road	09629850293	Stretford	Manchester	UK
2700	Schwanthalerstr. 7031	80925	Munich	Bavaria	DE
2800	Rua Frei Caneca 1360	01307-002	Sao Paulo	Sao Paulo	BR
2900	20 Rue des Corps-Saints	1730	Geneva	Geneve	CH
3000	Murtenstrasse 921	3095	Bern	BE	CH
3100	Pieter Breughelstraat 837	3029SK	Utrecht	Utrecht	NL
3200	Mariano Escobedo 9991	11932	Mexico City	Distrito Federal,	MX

23 rows selected.

DEPARTMENTS Table

```
DESCRIBE departments
```

Name	Null?	Type
DEPARTMENT_ID	NOT NULL	NUMBER(4)
DEPARTMENT_NAME	NOT NULL	VARCHAR2(30)
MANAGER_ID		NUMBER(6)
LOCATION_ID		NUMBER(4)

```
SELECT * FROM departments;
```

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
10	Administration	200	1700
20	Marketing	201	1800
30	Purchasing	114	1700
40	Human Resources	203	2400
50	Shipping	121	1500
60	IT	103	1400
70	Public Relations	204	2700
80	Sales	145	2500
90	Executive	100	1700
100	Finance	108	1700
110	Accounting	205	1700
120	Treasury		1700
130	Corporate Tax		1700
140	Control And Credit		1700
DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
150	Shareholder Services		1700
160	Benefits		1700
170	Manufacturing		1700
180	Construction		1700
190	Contracting		1700
200	Operations		1700
210	IT Support		1700
220	NOC		1700
230	IT Helpdesk		1700
240	Government Sales		1700
250	Retail Sales		1700
260	Recruiting		1700
270	Payroll		1700

27 rows selected.

JOBS Table

```
DESCRIBE jobs
```

Name	Null?	Type
JOB_ID	NOT NULL	VARCHAR2(10)
JOB_TITLE	NOT NULL	VARCHAR2(35)
MIN_SALARY		NUMBER(6)
MAX_SALARY		NUMBER(6)

```
SELECT * FROM jobs;
```

JOB_ID	JOB_TITLE	MIN_SALARY	MAX_SALARY
AD_PRES	President	20000	40000
AD_VP	Administration Vice President	15000	30000
AD_ASST	Administration Assistant	3000	6000
FI_MGR	Finance Manager	8200	16000
FI_ACCOUNT	Accountant	4200	9000
AC_MGR	Accounting Manager	8200	16000
AC_ACCOUNT	Public Accountant	4200	9000
SA_MAN	Sales Manager	10000	20000
SA_REP	Sales Representative	6000	12000
PU_MAN	Purchasing Manager	8000	15000
PU_CLERK	Purchasing Clerk	2500	5500
ST_MAN	Stock Manager	5500	8500
ST_CLERK	Stock Clerk	2000	5000
SH_CLERK	Shipping Clerk	2500	5500
JOB_ID	JOB_TITLE	MIN_SALARY	MAX_SALARY
IT_PROG	Programmer	4000	10000
MK_MAN	Marketing Manager	9000	15000
MK_REP	Marketing Representative	4000	9000
HR_REP	Human Resources Representative	4000	9000
PR_REP	Public Relations Representative	4500	10500

19 rows selected.

`EMPLOYEES` Table

```
DESCRIBE employees
```

Name	Null?	Type
EMPLOYEE_ID	NOT NULL	NUMBER(6)
FIRST_NAME		VARCHAR2(20)
LAST_NAME	NOT NULL	VARCHAR2(25)
EMAIL	NOT NULL	VARCHAR2(25)
PHONE_NUMBER		VARCHAR2(20)
HIRE_DATE	NOT NULL	DATE
JOB_ID	NOT NULL	VARCHAR2(10)
SALARY		NUMBER(8,2)
COMMISSION_PCT		NUMBER(2,2)
MANAGER_ID		NUMBER(6)
DEPARTMENT_ID		NUMBER(4)

EMPLOYEES Table

The headings for the COMMISSION_PCT, MANAGER_ID, and DEPARTMENT_ID columns are set to COMM, MGRID, and DEPTID in the following screenshot, to fit the table values across the page.

```
SELECT * FROM employees;
```

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	comm	mgrid	deptid
100	Steven	King	SKING	515.123.4567	17-JUN-87	AD_PRES	24000			90
101	Neena	Kochhar	NKOCHHAR	515.123.4568	21-SEP-89	AD_VP	17000		100	90
102	Lex	De Haan	LDEHAAN	515.123.4569	13-JAN-93	AD_VP	17000		100	90
103	Alexander	Hunold	AHUNOLD	590.423.4567	03-JAN-90	IT_PROG	9000		102	60
104	Bruce	Ernst	BERNST	590.423.4568	21-MAY-91	IT_PROG	6000		103	60
105	David	Austin	DAUSTIN	590.423.4569	25-JUN-97	IT_PROG	4800		103	60
106	Valli	Pataballa	VPATABAL	590.423.4560	05-FEB-98	IT_PROG	4800		103	60
107	Diana	Lorentz	DLORENTZ	590.423.5567	07-FEB-99	IT_PROG	4200		103	60
108	Nancy	Greenberg	NGREENBE	515.124.4569	17-AUG-94	FI_MGR	12000		101	100
109	Daniel	Faviet	DFAVIET	515.124.4169	16-AUG-94	FI_ACCOUNT	9000		108	100
110	John	Chen	JCHEN	515.124.4269	28-SEP-97	FI_ACCOUNT	8200		108	100
111	Ismael	Sciarra	ISCIARRA	515.124.4369	30-SEP-97	FI_ACCOUNT	7700		108	100
112	Jose Manuel	Urman	JMURMAN	515.124.4469	07-MAR-98	FI_ACCOUNT	7800		108	100
113	Luis	Popp	LPOPP	515.124.4567	07-DEC-99	FI_ACCOUNT	6900		108	100
EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	comm	mgrid	deptid
114	Den	Raphaely	DRAPHEAL	515.127.4561	07-DEC-94	PU_MAN	11000		100	30
115	Alexander	Khoo	AKHOO	515.127.4562	18-MAY-95	PU_CLERK	3100		114	30
116	Shelli	Baida	SBAIDA	515.127.4563	24-DEC-97	PU_CLERK	2900		114	30
117	Sigal	Tobias	STOBIAS	515.127.4564	24-JUL-97	PU_CLERK	2800		114	30
118	Guy	Himuro	GHIMURO	515.127.4565	15-NOV-98	PU_CLERK	2600		114	30
119	Karen	Colmenares	KCOLMENA	515.127.4566	10-AUG-99	PU_CLERK	2500		114	30
120	Matthew	Weiss	MWEISS	650.123.1234	18-JUL-96	ST_MAN	8000		100	50
121	Adam	Fripp	AFRIPP	650.123.2234	10-APR-97	ST_MAN	8200		100	50
122	Payam	Kaufling	PKAUFLIN	650.123.3234	01-MAY-95	ST_MAN	7900		100	50
123	Shanta	Vollman	SVOLLMAN	650.123.4234	10-OCT-97	ST_MAN	6500		100	50
124	Kevin	Mourgos	KMOURGOS	650.123.5234	16-NOV-99	ST_MAN	5800		100	50
125	Julia	Nayer	JNAYER	650.124.1214	16-JUL-97	ST_CLERK	3200		120	50
126	Irene	Mikkilineni	IMIKKILI	650.124.1224	28-SEP-98	ST_CLERK	2700		120	50
127	James	Landry	JLANDRY	650.124.1334	14-JAN-99	ST_CLERK	2400		120	50

`EMPLOYEES` Table (continued)

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	comm	mgrid	deptid
128	Steven	Markle	SMARKLE	650.124.1434	08-MAR-00	ST_CLERK	2200		120	50
129	Laura	Bissot	LBISSOT	650.124.5234	20-AUG-97	ST_CLERK	3300		121	50
130	Mozhe	Atkinson	MATKINSO	650.124.6234	30-OCT-97	ST_CLERK	2800		121	50
131	James	Marlow	JAMRLOW	650.124.7234	16-FEB-97	ST_CLERK	2500		121	50
132	TJ	Olson	TJOLSON	650.124.8234	10-APR-99	ST_CLERK	2100		121	50
133	Jason	Mallin	JMALLIN	650.127.1934	14-JUN-96	ST_CLERK	3300		122	50
134	Michael	Rogers	MROGERS	650.127.1834	26-AUG-98	ST_CLERK	2900		122	50
135	Ki	Gee	KGEE	650.127.1734	12-DEC-99	ST_CLERK	2400		122	50
136	Hazel	Philtanker	HPHILTAN	650.127.1634	06-FEB-00	ST_CLERK	2200		122	50
137	Renske	Ladwig	RLADWIG	650.121.1234	14-JUL-95	ST_CLERK	3600		123	50
138	Stephen	Stiles	SSTILES	650.121.2034	26-OCT-97	ST_CLERK	3200		123	50
139	John	Seo	JSEO	650.121.2019	12-FEB-98	ST_CLERK	2700		123	50
140	Joshua	Patel	JPATEL	650.121.1834	06-APR-98	ST_CLERK	2500		123	50
141	Trenna	Rajs	TRAJS	650.121.8009	17-OCT-95	ST_CLERK	3500		124	50
EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	comm	mgrid	deptid
142	Curtis	Davies	CDAVIES	650.121.2994	29-JAN-97	ST_CLERK	3100		124	50
143	Randall	Matos	RMATOS	650.121.2874	15-MAR-98	ST_CLERK	2600		124	50
144	Peter	Vargas	PVARGAS	650.121.2004	09-JUL-98	ST_CLERK	2500		124	50
145	John	Russell	JRUSSEL	011.44.1344.429268	01-OCT-96	SA_MAN	14000	.4	100	80
146	Karen	Partners	KPARTNER	011.44.1344.467268	05-JAN-97	SA_MAN	13500	.3	100	80
147	Alberto	Errazuriz	AERRAZUR	011.44.1344.429278	10-MAR-97	SA_MAN	12000	.3	100	80
148	Gerald	Cambrault	GCAMBRAU	011.44.1344.619268	15-OCT-99	SA_MAN	11000	.3	100	80
149	Eleni	Zlotkey	EZLOTKEY	011.44.1344.429018	29-JAN-00	SA_MAN	10500	.2	100	80
150	Peter	Tucker	PTUCKER	011.44.1344.129268	30-JAN-97	SA_REP	10000	.3	145	80
151	David	Bernstein	DBERNSTE	011.44.1344.345268	24-MAR-97	SA_REP	9500	.25	145	80
152	Peter	Hall	PHALL	011.44.1344.478968	20-AUG-97	SA_REP	9000	.25	145	80
153	Christopher	Olsen	COLSEN	011.44.1344.498718	30-MAR-98	SA_REP	8000	.2	145	80
154	Nanette	Cambrault	NCAMBRAU	011.44.1344.987668	09-DEC-98	SA_REP	7500	.2	145	80
155	Oliver	Tuvault	OTUVAULT	011.44.1344.486508	23-NOV-99	SA_REP	7000	.15	145	80
EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	comm	mgrid	deptid
156	Janette	King	JKING	011.44.1345.429268	30-JAN-96	SA_REP	10000	.35	146	80
157	Patrick	Sully	PSULLY	011.44.1345.929268	04-MAR-96	SA_REP	9500	.35	146	80
158	Allan	McEwen	AMCEWEN	011.44.1345.829268	01-AUG-96	SA_REP	9000	.35	146	80
159	Lindsey	Smith	LSMITH	011.44.1345.729268	10-MAR-97	SA_REP	8000	.3	146	80
160	Louise	Doran	LDORAN	011.44.1345.629268	15-DEC-97	SA_REP	7500	.3	146	80
161	Sarath	Sewall	SSEWALL	011.44.1345.529268	03-NOV-98	SA_REP	7000	.25	146	80
162	Clara	Vishney	CVISHNEY	011.44.1346.129268	11-NOV-97	SA_REP	10500	.25	147	80
163	Danielle	Greene	DGREENE	011.44.1346.229268	19-MAR-99	SA_REP	9500	.15	147	80
164	Mattea	Marvins	MMARVINS	011.44.1346.329268	24-JAN-00	SA_REP	7200	.1	147	80
165	David	Lee	DLEE	011.44.1346.529268	23-FEB-00	SA_REP	6800	.1	147	80
166	Sundar	Ande	SANDE	011.44.1346.629268	24-MAR-00	SA_REP	6400	.1	147	80
167	Amit	Banda	ABANDA	011.44.1346.729268	21-APR-00	SA_REP	6200	.1	147	80
168	Lisa	Ozer	LOZER	011.44.1343.929268	11-MAR-97	SA_REP	11500	.25	148	80
169	Harrison	Bloom	HBLOOM	011.44.1343.829268	23-MAR-98	SA_REP	10000	.2	148	80

EMPLOYEES Table (continued)

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	comm	mgrid	deptid
170	Tayler	Fox	TFOX	011.44.1343.729268	24-JAN-98	SA_REP	9600	.2	148	80
171	William	Smith	WSMITH	011.44.1343.629268	23-FEB-99	SA_REP	7400	.15	148	80
172	Elizabeth	Bates	EBATES	011.44.1343.529268	24-MAR-99	SA_REP	7300	.15	148	80
173	Sundita	Kumar	SKUMAR	011.44.1343.329268	21-APR-00	SA_REP	6100	.1	148	80
174	Ellen	Abel	EABEL	011.44.1644.429267	11-MAY-96	SA_REP	11000	.3	149	80
175	Alyssa	Hutton	AHUTTON	011.44.1644.429266	19-MAR-97	SA_REP	8800	.25	149	80
176	Jonathon	Taylor	JTAYLOR	011.44.1644.429265	24-MAR-98	SA_REP	8600	.2	149	80
177	Jack	Livingston	JLIVINGS	011.44.1644.429264	23-APR-98	SA_REP	8400	.2	149	80
178	Kimberely	Grant	KGRANT	011.44.1644.429263	24-MAY-99	SA_REP	7000	.15	149	
179	Charles	Johnson	CJOHNSON	011.44.1644.429262	04-JAN-00	SA_REP	6200	.1	149	80
180	Winston	Taylor	WTAYLOR	650.507.9876	24-JAN-98	SH_CLERK	3200		120	50
181	Jean	Fleaur	JFLEAUR	650.507.9877	23-FEB-98	SH_CLERK	3100		120	50
182	Martha	Sullivan	MSULLIVA	650.507.9878	21-JUN-99	SH_CLERK	2500		120	50
183	Girard	Geoni	GGEONI	650.507.9879	03-FEB-00	SH_CLERK	2800		120	50
EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	comm	mgrid	deptid
184	Nandita	Sarchand	NSARCHAN	650.509.1876	27-JAN-96	SH_CLERK	4200		121	50
185	Alexis	Bull	ABULL	650.509.2876	20-FEB-97	SH_CLERK	4100		121	50
186	Julia	Dellinger	JDELLING	650.509.3876	24-JUN-98	SH_CLERK	3400		121	50
187	Anthony	Cabrio	ACABRIO	650.509.4876	07-FEB-99	SH_CLERK	3000		121	50
188	Kelly	Chung	KCHUNG	650.505.1876	14-JUN-97	SH_CLERK	3800		122	50
189	Jennifer	Dilly	JDILLY	650.505.2876	13-AUG-97	SH_CLERK	3600		122	50
190	Timothy	Gates	TGATES	650.505.3876	11-JUL-98	SH_CLERK	2900		122	50
191	Randall	Perkins	RPERKINS	650.505.4876	19-DEC-99	SH_CLERK	2500		122	50
192	Sarah	Bell	SBELL	650.501.1876	04-FEB-96	SH_CLERK	4000		123	50
193	Britney	Everett	BEVERETT	650.501.2876	03-MAR-97	SH_CLERK	3900		123	50
194	Samuel	McCain	SMCCAIN	650.501.3876	01-JUL-98	SH_CLERK	3200		123	50
195	Vance	Jones	VJONES	650.501.4876	17-MAR-99	SH_CLERK	2800		123	50
196	Alana	Walsh	AWALSH	650.507.9811	24-APR-98	SH_CLERK	3100		124	50
197	Kevin	Feeney	KFEENEY	650.507.9822	23-MAY-98	SH_CLERK	3000		124	50
EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	comm	mgrid	deptid
198	Donald	OConnell	DOCONNEL	650.507.9833	21-JUN-99	SH_CLERK	2600		124	50
199	Douglas	Grant	DGRANT	650.507.9844	13-JAN-00	SH_CLERK	2600		124	50
200	Jennifer	Whalen	JWHALEN	515.123.4444	17-SEP-87	AD_ASST	4400		101	10
201	Michael	Hartstein	MHARTSTE	515.123.5555	17-FEB-96	MK_MAN	13000		100	20
202	Pat	Fay	PFAY	603.123.6666	17-AUG-97	MK_REP	6000		201	20
203	Susan	Mavris	SMAVRIS	515.123.7777	07-JUN-94	HR_REP	6500		101	40
204	Hermann	Baer	HBAER	515.123.8888	07-JUN-94	PR_REP	10000		101	70
205	Shelley	Higgins	SHIGGINS	515.123.8080	07-JUN-94	AC_MGR	12000		101	110
206	William	Gietz	WGIETZ	515.123.8181	07-JUN-94	AC_ACCOUNT	8300		205	110

107 rows selected.

`JOB_HISTORY` **Table**

```
DESCRIBE job_history
```

Name	Null?	Type
EMPLOYEE_ID	NOT NULL	NUMBER(6)
START_DATE	NOT NULL	DATE
END_DATE	NOT NULL	DATE
JOB_ID	NOT NULL	VARCHAR2(10)
DEPARTMENT_ID		NUMBER(4)

```
SELECT * FROM job_history;
```

EMPLOYEE_ID	START_DAT	END_DATE	JOB_ID	deptid
102	13-JAN-93	24-JUL-98	IT_PROG	60
101	21-SEP-89	27-OCT-93	AC_ACCOUNT	110
101	28-OCT-93	15-MAR-97	AC_MGR	110
201	17-FEB-96	19-DEC-99	MK_REP	20
114	24-MAR-98	31-DEC-99	ST_CLERK	50
122	01-JAN-99	31-DEC-99	ST_CLERK	50
200	17-SEP-87	17-JUN-93	AD_ASST	90
176	24-MAR-98	31-DEC-98	SA_REP	80
176	01-JAN-99	31-DEC-99	SA_MAN	80
200	01-JUL-94	31-DEC-98	AC_ACCOUNT	90

10 rows selected.

Studies for Implementing Triggers

Objectives

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

- **Enhance database security with triggers**
- **Audit data changes using DML triggers**
- **Enforce data integrity with DML triggers**
- **Maintain referential integrity using triggers**
- **Use triggers to replicate data between tables**
- **Use triggers to automate computation of derived data**
- **Provide event-logging capabilities using triggers**

Lesson Aim

In this lesson, you learn to develop database triggers in order to enhance features that cannot otherwise be implemented by the Oracle server. In some cases, it may be sufficient to refrain from using triggers and accept the functionality provided by the Oracle server.

This lesson covers the following business application scenarios:

- Security
- Auditing
- Data integrity
- Referential integrity
- Table replication
- Computing derived data automatically
- Event logging

Controlling Security Within the Server

Using database security with the `GRANT` statement.

```
GRANT SELECT, INSERT, UPDATE, DELETE
 ON    employees
 TO    clerk;              -- database role
GRANT clerk TO scott;
```


Controlling Security Within the Server

Develop schemas and roles within the Oracle server to control the security of data operations on tables according to the identity of the user.

- Base privileges upon the username supplied when the user connects to the database.
- Determine access to tables, views, synonyms, and sequences.
- Determine query, data-manipulation, and data-definition privileges.

Controlling Security with a Database Trigger

```
CREATE OR REPLACE TRIGGER secure_emp
  BEFORE INSERT OR UPDATE OR DELETE ON employees
DECLARE
dummy PLS_INTEGER;
BEGIN
 IF (TO_CHAR (SYSDATE, 'DY') IN ('SAT','SUN')) THEN
   RAISE_APPLICATION_ERROR(-20506,'You may only
     change data during normal business hours.');
 END IF;
 SELECT COUNT(*) INTO dummy FROM holiday
 WHERE holiday_date = TRUNC (SYSDATE);
 IF dummy > 0 THEN
   RAISE_APPLICATION_ERROR(-20507,
     'You may not change data on a holiday.');
 END IF;
END;
/
```

ORACLE

Controlling Security with a Database Trigger

Develop triggers to handle more complex security requirements.

- Base privileges on any database values, such as the time of day, the day of the week, and so on.
- Determine access to tables only.
- Determine data-manipulation privileges only.

Using the Server Facility to Audit Data Operations

The Oracle server stores the audit information in a data dictionary table or an operating system file.

```
AUDIT INSERT, UPDATE, DELETE
  ON  departments
  BY ACCESS
WHENEVER SUCCESSFUL;
```

```
Audit succeeded.
```


Auditing Data Operations

You can audit data operations within the Oracle server. Database auditing is used to monitor and gather data about specific database activities. The DBA can gather statistics such as which tables are being updated, how many I/Os are performed, how many concurrent users connect at peak time, and so on.

- Audit users, statements, or objects.
- Audit data retrieval, data-manipulation, and data-definition statements.
- Write the audit trail to a centralized audit table.
- Generate audit records once per session or once per access attempt.
- Capture successful attempts, unsuccessful attempts, or both.
- Enable and disable dynamically.

Executing SQL through PL/SQL program units may generate several audit records because the program units may refer to other database objects.

Auditing by Using a Trigger

```
CREATE OR REPLACE TRIGGER audit_emp_values
AFTER DELETE OR INSERT OR UPDATE
ON employees FOR EACH ROW
BEGIN
 IF (audit_emp_pkg. reason IS NULL) THEN
   RAISE_APPLICATION_ERROR (-20059, 'Specify a
    reason for operation through the procedure
    AUDIT_EMP_PKG.SET_REASON to proceed.');
 ELSE
   INSERT INTO audit_emp_table (user_name,
     timestamp, id, old_last_name, new_last_name,
     old_salary, new_salary, comments)
   VALUES (USER, SYSDATE, :OLD.employee_id,
     :OLD.last_name, :NEW.last_name,:OLD.salary,
     :NEW.salary, audit_emp_pkg.reason);
 END IF;
END;
```

```
CREATE OR REPLACE TRIGGER cleanup_audit_emp
 AFTER INSERT OR UPDATE OR DELETE ON employees
BEGIN audit_emp_package.g_reason := NULL;
END;
```

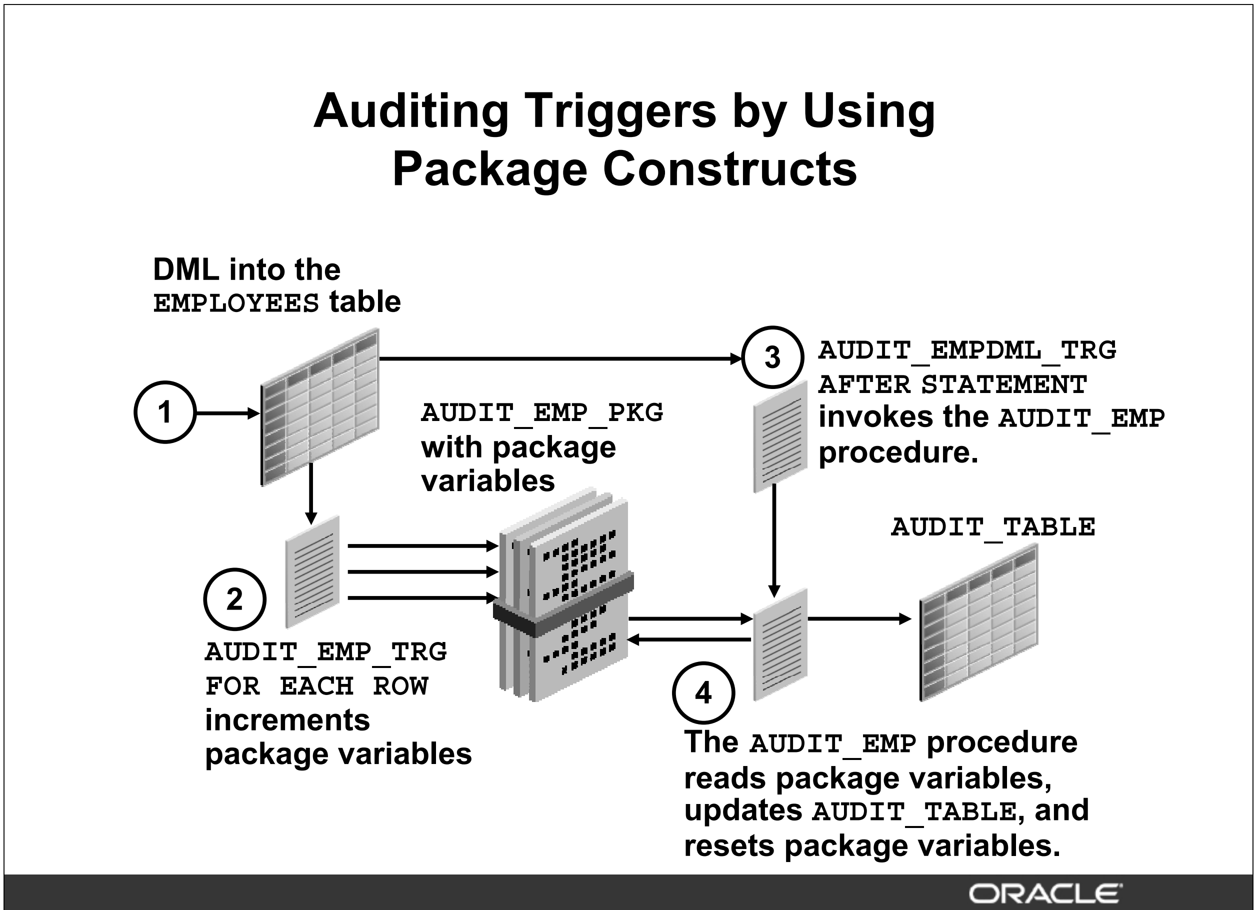

Auditing Data Values

Audit actual data values with triggers.

You can do the following:

- Audit data-manipulation statements only.
- Write the audit trail to a user-defined audit table.
- Generate audit records once for the statement or once for each row.
- Capture successful attempts only.
- Enable and disable dynamically.

Using the Oracle server, you can perform database auditing. Database auditing cannot record changes to specific column values. If the changes to the table columns need to be tracked and column values need to be stored for each change, then use application auditing. Application auditing can be done either through stored procedures or database triggers, as shown in the example in the slide.



Auditing Triggers by Using Package Constructs

The following pages cover PL/SQL subprograms with examples of the interaction of triggers, package procedures, functions, and global variables.

The sequence of events:

1. Execute an `INSERT`, `UPDATE`, or `DELETE` command that can manipulate one or many rows.
2. `AUDIT_EMP_TRG` (the `AFTER ROW` trigger) calls the package procedure to increment the global variables in the `VAR_PACK` package. Because this is a row trigger, the trigger fires once for each row that you updated.
3. When the statement has finished, `AUDIT_EMP_TAB` (the `AFTER STATEMENT` trigger) calls the `AUDIT_EMP` procedure.
4. This procedure assigns the values of the global variables into local variables using the package functions, updates the `AUDIT_TABLE`, and then resets the global variables.

Auditing Triggers by Using Package Constructs

`AFTER` statement trigger:

```
CREATE OR REPLACE TRIGGER audit_empdml_trg
AFTER UPDATE OR INSERT OR DELETE on employees
BEGIN
  audit_emp;      -- write the audit data
END audit_emp_tab;
/
```

`AFTER` row trigger:

```
CREATE OR REPLACE TRIGGER audit_emp_trg
AFTER UPDATE OR INSERT OR DELETE ON EMPLOYEES
FOR EACH ROW
-- Call Audit package to maintain counts
CALL audit_emp_pkg.set(INSERTING,UPDATING,DELETING);
/
```


Auditing Triggers by Using Package Constructs (continued)

The AUDIT_EMP_TRIG trigger is a row trigger that fires after every row is manipulated. This trigger invokes the package procedures depending on the type of data manipulation language (DML) performed. For example, if the DML updates the salary of an employee, then the trigger invokes the SET_G_UP_SAL procedure. This package procedure, in turn, invokes the G_UP_SAL function. This function increments the GV_UP_SAL package variable that keeps account of the number of rows being changed due to the update of the salary.

The AUDIT_EMP_TAB trigger fires after the statement has finished. This trigger invokes the AUDIT_EMP procedure, which is explained on the following pages. The AUDIT_EMP procedure updates the AUDIT_TABLE table. An entry is made into the AUDIT_TABLE table with information such as the user who performed the DML, the table on which DML is performed, and the total number of such data manipulations performed so far on the table (indicated by the value of the corresponding column in the AUDIT_TABLE table). At the end, the AUDIT_EMP procedure resets the package variables to 0.

AUDIT_PKG Package

```
CREATE OR REPLACE PACKAGE audit_emp_pkg IS
  delcnt PLS_INTEGER := 0;
  inscnt PLS_INTEGER := 0;
  updcnt  PLS_INTEGER := 0;
  PROCEDURE init;
  PROCEDURE set(i BOOLEAN,u BOOLEAN,d BOOLEAN);
END audit_emp_pkg;
/
CREATE OR REPLACE PACKAGE BODY audit_emp_pkg IS
 PROCEDURE init IS
  BEGIN
    inscnt := 0; updcnt := 0; delcnt := 0;
  END;
  PROCEDURE set(i BOOLEAN,u BOOLEAN,d BOOLEAN) IS
  BEGIN
    IF i THEN inscnt := inscnt + 1;
    ELSIF d THEN delcnt := delcnt + 1;
    ELSE upd := updcnt + 1;
    END IF;
  END;
END audit_emp_pkg;
/
```

ORACLE

AUDIT_PKG Package

The AUDIT_PKG package declares public package variables (inscnt, updcnt, and delcnt) that are used to track the number of INSERT, UPDATE, and DELETE operations performed. In the code example, they are declared publicly for simplicity. However, it may be better to declare them as private variables to prevent them from being directly modified. If the variables are declared privately, in the package body, you would have to provide additional public subprograms to return their values to the user of the package.

The init procedure is used to initialize the public package variables to zero.

The set procedure accepts three BOOLEAN arguments: i, u, and d for an INSERT, UPDATE, or DELETE operation, respectively. The appropriate parameter value is set to TRUE when the trigger that invokes the set procedure is fired during one of the DML operations. A package variable is incremented by a value of 1, depending on which argument value is TRUE when the set procedure is invoked.

Note: A DML trigger can fire once for each DML on each row. Therefore, only one of the three variables passed to the set procedure can be TRUE at a given time. The remaining two arguments will be set to the value FALSE.

AUDIT_TABLE Table and AUDIT_EMP Procedure

```
CREATE TABLE audit_table (
 USER_NAME   VARCHAR2(30),
 TABLE_NAME  VARCHAR2(30),
 INS         NUMBER,
 UPD         NUMBER,
 DEL         NUMBER)
/
CREATE OR REPLACE PROCEDURE audit_emp IS
BEGIN
  IF  delcnt + inscnt + updcnt <> 0 THEN
    UPDATE audit_table
     SET del = del + audit_emp_pkg.delcnt,
         ins = ins + audit_emp_pkg.inscnt,
         upd = upd + audit_emp_pkg.updcnt
    WHERE user_name = USER
    AND   table_name = 'EMPLOYEES';
    audit_emp_pkg.init;
  END IF;
END audit_emp;
/
```


AUDIT_TABLE Table and AUDIT_EMP Procedure

The AUDIT_EMP procedure updates the AUDIT_TABLE table and calls the functions in the AUDIT_EMP_PKG package that reset the package variables, ready for the next DML statement.

Enforcing Data Integrity Within the Server

```
ALTER TABLE employees ADD
  CONSTRAINT ck_salary CHECK (salary >= 500);
```

```
Table altered.
```


Enforcing Data Integrity Within the Server

You can enforce data integrity within the Oracle server and develop triggers to handle more complex data integrity rules.

The standard data integrity rules are not null, unique, primary key, and foreign key.

Use these rules to:

- Provide constant default values
- Enforce static constraints
- Enable and disable dynamically

Example

The code sample in the slide ensures that the salary is at least $500.

Protecting Data Integrity with a Trigger

```
CREATE OR REPLACE TRIGGER check_salary
  BEFORE UPDATE OF salary ON employees
  FOR EACH ROW
  WHEN (NEW.salary < OLD.salary)
BEGIN
  RAISE_APPLICATION_ERROR (-20508,
      'Do not decrease salary.');
END;
/
```

ORACLE

Protecting Data Integrity with a Trigger

Protect data integrity with a trigger and enforce nonstandard data integrity checks.

- Provide variable default values.
- Enforce dynamic constraints.
- Enable and disable dynamically.
- Incorporate declarative constraints within the definition of a table to protect data integrity.

Example

The code sample in the slide ensures that the salary is never decreased.

Enforcing Referential Integrity Within the Server

```
ALTER TABLE employees
  ADD CONSTRAINT emp_deptno_fk
  FOREIGN KEY (department_id)
     REFERENCES departments(department_id)
ON DELETE CASCADE;
```


Enforcing Referential Integrity Within the Server

Incorporate referential integrity constraints within the definition of a table to prevent data inconsistency and enforce referential integrity within the server.

- Restrict updates and deletes.
- Cascade deletes.
- Enable and disable dynamically.

Example

When a department is removed from the DEPARTMENTS parent table, cascade the deletion to the corresponding rows in the EMPLOYEES child table.

Protecting Referential Integrity with a Trigger

```
CREATE OR REPLACE TRIGGER cascade_updates
 AFTER UPDATE OF department_id ON departments
 FOR EACH ROW
BEGIN
 UPDATE employees
  SET employees.department_id=:NEW.department_id
  WHERE employees.department_id=:OLD.department_id;
 UPDATE job_history
  SET department_id=:NEW.department_id
  WHERE department_id=:OLD.department_id;
END;
/
```

ORACLE

Protecting Referential Integrity with a Trigger

The following referential integrity rules are not supported by declarative constraints:

- Cascade updates.
- Set to NULL for updates and deletions.
- Set to a default value on updates and deletions.
- Enforce referential integrity in a distributed system.
- Enable and disable dynamically.

You can develop triggers to implement these integrity rules.

Example

Enforce referential integrity with a trigger. When the value of DEPARTMENT_ID changes in the DEPARTMENTS parent table, cascade the update to the corresponding rows in the EMPLOYEES child table.

For a complete referential integrity solution using triggers, a single trigger is not enough.

Replicating a Table Within the Server

```
CREATE MATERIALIZED VIEW emp_copy
  NEXT sysdate + 7
  AS SELECT * FROM employees@ny;
```


Creating a Materialized View

Materialized views enable you to maintain copies of remote data on your local node for replication purposes. You can select data from a materialized view as you would from a normal database table or view. A materialized view is a database object that contains the results of a query, or a copy of some database on a query. The `FROM` clause of the query of a materialized view can name tables, views, and other materialized views.

When a materialized view is used, replication is performed implicitly by the Oracle server. This performs better than using user-defined PL/SQL triggers for replication. Materialized views:

- Copy data from local and remote tables asynchronously, at user-defined intervals
- Can be based on multiple master tables
- Are read-only by default, unless using the Oracle Advanced Replication feature
- Improve the performance of data manipulation on the master table

Alternatively, you can replicate tables using triggers.

The example in the slide creates a copy of the remote `EMPLOYEES` table from New York. The `NEXT` clause specifies a date time expression for the interval between automatic refreshes.

Replicating a Table with a Trigger

```
CREATE OR REPLACE TRIGGER emp_replica
 BEFORE INSERT OR UPDATE ON employees FOR EACH ROW
BEGIN /* Proceed if user initiates data operation,
        NOT through the cascading trigger.*/
  IF INSERTING THEN
   IF :NEW.flag IS NULL THEN
     INSERT INTO employees@sf
     VALUES(:new.employee_id,...,'B');
     :NEW.flag := 'A';
   END IF;
  ELSE   /* Updating. */
   IF :NEW.flag = :OLD.flag THEN
     UPDATE employees@sf
      SET ename=:NEW.last_name,...,flag=:NEW.flag
      WHERE employee_id = :NEW.employee_id;
   END IF;
   IF :OLD.flag = 'A' THEN :NEW.flag := 'B';
                      ELSE :NEW.flag := 'A';
   END IF;
  END IF;
END;
```


Replicating a Table with a Trigger

You can replicate a table with a trigger. By replicating a table, you can:

- Copy tables synchronously, in real time
- Base replicas on a single master table
- Read from replicas as well as write to them

Note: Excessive use of triggers can impair the performance of data manipulation on the master table, particularly if the network fails.

Example

In New York, replicate the local `EMPLOYEES` table to San Francisco.

Computing Derived Data Within the Server

```
UPDATE departments
 SET total_sal=(SELECT SUM(salary)
                FROM employees
                WHERE employees.department_id =
                     departments.department_id);
```


Computing Derived Data Within the Server

By using the server, you can schedule batch jobs or use the database Scheduler for the following scenarios:

- Compute derived column values asynchronously, at user-defined intervals.
- Store derived values only within database tables.
- Modify data in one pass to the database and calculate derived data in a second pass.

Alternatively, you can use triggers to keep running computations of derived data.

Example

Keep the salary total for each department within a special `TOTAL_SALARY` column of the `DEPARTMENTS` table.

Computing Derived Values with a Trigger

```
CREATE PROCEDURE increment_salary
  (id NUMBER, new_sal NUMBER) IS
BEGIN
   UPDATE departments
   SET    total_sal = NVL (total_sal, 0)+ new_sal
   WHERE  department_id = id;
END increment_salary;
```

```
CREATE OR REPLACE TRIGGER compute_salary
AFTER INSERT OR UPDATE OF salary OR DELETE
ON employees FOR EACH ROW
BEGIN
 IF DELETING THEN     increment_salary(
     :OLD.department_id,(-1*:OLD.salary));
 ELSIF UPDATING THEN  increment_salary(
     :NEW.department_id,(:NEW.salary-:OLD.salary));
 ELSE   increment_salary(
     :NEW.department_id,:NEW.salary); --INSERT
 END IF;
END;
```


Computing Derived Data Values with a Trigger

By using a trigger, you can perform the following tasks:

- Compute derived columns synchronously, in real time.
- Store derived values within database tables or within package global variables.
- Modify data and calculate derived data in a single pass to the database.

Example

Keep a running total of the salary for each department in the special TOTAL_SALARY column of the DEPARTMENTS table.

Logging Events with a Trigger

```
CREATE OR REPLACE TRIGGER notify_reorder_rep
BEFORE UPDATE OF quantity_on_hand, reorder_point
 ON inventories FOR EACH ROW
DECLARE
 dsc product_descriptions.product_description%TYPE;
 msg_text VARCHAR2(2000);
BEGIN
  IF :NEW.quantity_on_hand <=
     :NEW.reorder_point THEN
    SELECT product_description INTO dsc
    FROM product_descriptions
    WHERE product_id = :NEW.product_id;
  _ msg_text := 'ALERT: INVENTORY LOW ORDER:'||
       'Yours,' ||CHR(10) ||user || '.'|| CHR(10);
  ELSIF :OLD.quantity_on_hand >=
        :NEW.quantity_on_hand THEN
    msg_text := 'Product #'||... CHR(10);
  END IF;
  UTL_MAIL.SEND('inv@oracle.com','ord@oracle.com',
   message=>msg_text, subject=>'Inventory Notice');
END;
```


Logging Events with a Trigger

In the server, you can log events by querying data and performing operations manually. This sends an e-mail message when the inventory for a particular product has fallen below the acceptable limit. This trigger uses the Oracle-supplied package `UTL_MAIL` to send the e-mail message.

Logging Events Within the Server

1. Query data explicitly to determine whether an operation is necessary.
2. Perform the operation, such as sending a message.

Using Triggers to Log Events

1. Perform operations implicitly, such as firing off an automatic electronic memo.
2. Modify data and perform its dependent operation in a single step.
3. Log events automatically as data is changing.

Logging Events with a Trigger (continued)

Logging Events Transparently

In the trigger code:

- `CHR(10)` is a carriage return
- `Reorder_point` is not `NULL`
- Another transaction can receive and read the message in the pipe

Example

```
 CREATE OR REPLACE TRIGGER notify_reorder_rep
 BEFORE UPDATE OF amount_in_stock, reorder_point
 ON inventory  FOR EACH ROW
 DECLARE
   dsc product.descrip%TYPE;
   msg_text VARCHAR2(2000);
 BEGIN
   IF  :NEW.amount_in_stock <= :NEW.reorder_point THEN
     SELECT descrip INTO  dsc
     FROM PRODUCT WHERE prodid = :NEW.product_id;
     msg_text := 'ALERT: INVENTORY LOW ORDER:'||CHR(10)||
     'It has come to my personal attention that, due to recent'
     ||CHR(10)||'transactions, our inventory for product # '||
     TO_CHAR(:NEW.product_id)||'-- '|| dsc ||
     ' -- has fallen below acceptable levels.' || CHR(10) ||
     'Yours,' ||CHR(10) ||user || '.'|| CHR(10)|| CHR(10);
   ELSIF :OLD.amount_in_stock >= :NEW.amount_in_stock THEN
     msg_text := 'Product #'|| TO_CHAR(:NEW.product_id)
     ||' ordered. '|| CHR(10)|| CHR(10);
   END IF;
   UTL_MAIL.SEND('inv@oracle.com', 'ord@oracle.com',
      message => msg_text, subject => 'Inventory Notice');
END;
```

Summary

In this lesson, you should have learned how to:

- Use database triggers and database server functionality to:
  - Enhance database security
  - Audit data changes
  - Enforce data integrity
  - Maintain referential integrity
  - Replicate data between tables
  - Automate computation of derived data
  - Provide event-logging capabilities
- Recognize when to use triggers to database functionality

Summary

This lesson provides some detailed comparison of using the Oracle database server functionality to implement security, auditing, data integrity, replication, and logging. The lesson also covers how database triggers can be used to implement the same features but go further to enhance the features that the database server provides. In some cases, you must use a trigger to perform some activities (such as computation of derived data) because the Oracle server cannot know how to implement this kind of business rule without some programming effort.

Review of PL/SQL

Block Structure for Anonymous PL/SQL Blocks

- `DECLARE` (optional)
  - Declare PL/SQL objects to be used within this block.
- `BEGIN` (mandatory)
  - Define the executable statements.
- `EXCEPTION` (optional)
  - Define the actions that take place if an error or exception arises.
- `END;` (mandatory)

Anonymous Blocks

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

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

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

Declaring PL/SQL Variables

- **Syntax:**

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

- **Examples:**

```
Declare
  v_hiredate       DATE;
  v_deptno         NUMBER(2) NOT NULL := 10;
  v_location       VARCHAR2(13) := 'Atlanta';
  c_ comm          CONSTANT NUMBER := 1400;
  v_count          BINARY_INTEGER := 0;
  v_valid          BOOLEAN NOT NULL := TRUE;
```

ORACLE

Declaring PL/SQL Variables

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

In the syntax,

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

Declaring Variables with the %TYPE Attribute

Examples:

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


Declaring Variables with the %TYPE Attribute

Declare variables to store the name of an employee.

```
...
v_ename          employees.last_name%TYPE;
...
```

Declare variables to store the balance of a bank account, as well as the minimum balance, which starts out as 10.

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

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

Creating a PL/SQL Record

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

Example:

```
...
  TYPE emp_record_type IS RECORD
    (ename   VARCHAR2(25),
     job        VARCHAR2(10),
     sal        NUMBER(8,2));
  emp_record    emp_record_type;
...
```


Creating a PL/SQL Record

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

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

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

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

%ROWTYPE Attribute

Examples:

- **Declare a variable to store the same information about a department as is stored in the DEPARTMENTS table.**

```
dept_record    departments%ROWTYPE;
```

- **Declare a variable to store the same information about an employee as is stored in the EMPLOYEES table.**

```
emp_record     employees%ROWTYPE;
```


Examples

The first declaration in the slide creates a record with the same field names and field data types as a row in the DEPARTMENTS table. The fields are DEPARTMENT_ID, DEPARTMENT_NAME, MANAGER_ID, and LOCATION_ID.

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

In the following example, you select column values into a record named item_record.

```
DECLARE
   job_record  jobs%ROWTYPE;
   ...
BEGIN
   SELECT * INTO job_record
   FROM   jobs
   WHERE  ...
```

Creating a PL/SQL Table

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

ORACLE

Creating a PL/SQL Table

There are no predefined data types for PL/SQL tables, as there are for scalar variables. Therefore, you must create the data type first and then declare an identifier using that data type.

Referencing a PL/SQL Table

Syntax

```
pl/sql_table_name(primary_key_value)
```

In this syntax, `primary_key_value` belongs to the `BINARY_INTEGER` type.

Reference the third row in a PL/SQL table `ENAME_TABLE`.

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

The magnitude range of a `BINARY_INTEGER` is –2147483647 to 2147483647. The primary key value can therefore be negative. Indexing need not start with 1.

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

SELECT Statements in PL/SQL

The INTO clause is mandatory.

Example:

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


INTO Clause

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

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

Queries Must Return One and Only One Row

SELECT statements within a PL/SQL block fall into the ANSI classification of Embedded SQL, for which the following rule applies:

Queries must return one and only one row. More than one row or no row generates an error.

PL/SQL deals with these errors by raising standard exceptions, which you can trap in the exception section of the block with the NO_DATA_FOUND and TOO_MANY_ROWS exceptions. You should code SELECT statements to return a single row.

Inserting Data

Add new employee information to the `EMPLOYEES` table.

Example:

```
DECLARE
  v_empid  employees.employee_id%TYPE;
BEGIN
  SELECT  employees_seq.NEXTVAL
  INTO    v_empno
  FROM    dual;
  INSERT INTO  employees(employee_id, last_name,
                         job_id, department_id)
  VALUES(v_empno, 'HARDING', 'PU_CLERK', 30);
END;
```


Inserting Data

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

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

Updating Data

Increase the salary of all employees in the `EMPLOYEES` table who are purchasing clerks.

Example:

```
DECLARE
  v_sal_increase   employees.salary%TYPE := 2000;
BEGIN
  UPDATE  employees
  SET     salary = salary + v_sal_increase
  WHERE   job_id = 'PU_CLERK';
END;
```


Updating Data

There may be ambiguity in the `SET` clause of the `UPDATE` statement because, although the identifier on the left of the assignment operator is always a database column, the identifier on the right can be either a database column or a PL/SQL variable.

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

Note: PL/SQL variable assignments always use `:=` and SQL column assignments always use `=`. Remember that if column names and identifier names are identical in the `WHERE` clause, the Oracle server looks to the database first for the name.

Deleting Data

Delete rows that belong to department 190 from the `EMPLOYEES` table.

Example:

```
DECLARE
  v_deptid   employees.department_id%TYPE := 190;
BEGIN
  DELETE FROM employees
  WHERE department_id = v_deptid;
END;
```


Deleting Data

Delete a specific job:

```
DECLARE
  v_jobid    jobs.job_id%TYPE := 'PR_REP';
BEGIN
  DELETE FROM jobs
  WHERE job_id = v_jobid;
END;
```

COMMIT and ROLLBACK Statements

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

Controlling Transactions

You control the logic of transactions with COMMIT and ROLLBACK SQL statements, rendering some groups of database changes permanent while discarding others. As with the Oracle server, data manipulation language (DML) transactions start at the first command to follow a COMMIT or ROLLBACK and end on the next successful COMMIT or ROLLBACK. These actions may occur within a PL/SQL block or as a result of events in the host environment. A COMMIT ends the current transaction by making all pending changes to the database permanent.

Syntax

```
COMMIT [WORK];
ROLLBACK [WORK];
```

In this syntax, WORK is for compliance with ANSI standards.

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

You can also include explicit locking commands (such as LOCK TABLE and SELECT ... FOR UPDATE) in a block. They stay in effect until the end of the transaction. Also, one PL/SQL block does not necessarily imply one transaction.

SQL Cursor Attributes

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

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

ORACLE

SQL Cursor Attributes

SQL cursor attributes enable you to evaluate what happened when the implicit cursor was last used. You use these attributes in PL/SQL statements such as functions. You cannot use them in SQL statements.

You can use the `SQL%ROWCOUNT`, `SQL%FOUND`, `SQL%NOTFOUND`, and `SQL%ISOPEN` attributes in the exception section of a block to gather information about the execution of a DML statement. In PL/SQL, a DML statement that does not change any rows is not seen as an error condition, whereas the `SELECT` statement will return an exception if it cannot locate any rows.

IF, THEN, and ELSIF Statements

For a given value entered, return a calculated value.

Example:

```
. . .
IF v_start > 100 THEN
  v_start := 2 * v_start;
ELSIF v_start >= 50 THEN
  v_start := 0.5 * v_start;
ELSE
  v_start := 0.1 * v_start;
END IF;
. . .
```

ORACLE

IF, THEN, and ELSIF Statements

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

Example

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

The statement in the slide is further defined as follows:

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

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

Basic Loop

Example:

```
DECLARE
  v_ordid     order_items.order_id%TYPE := 101;
  v_counter   NUMBER(2) := 1;
BEGIN
  LOOP
    INSERT INTO order_items(order_id,line_item_id)
    VALUES(v_ordid, v_counter);
    v_counter := v_counter + 1;
    EXIT WHEN v_counter > 10;
  END LOOP;
END;
```


Basic Loop

The basic loop example shown in the slide is defined as follows:

Insert the first 10 new line items for order number 101.

Note: A basic loop enables execution of its statements at least once, even if the condition has been met upon entering the loop.

FOR Loop

Insert the first 10 new line items for order number 101.

Example:

```
DECLARE
  v_ordid     order_items.order_id%TYPE := 101;
BEGIN
  FOR i IN 1..10 LOOP
    INSERT INTO order_items(order_id,line_item_id)
    VALUES(v_ordid, i);
  END LOOP;
END;
```

ORACLE

FOR Loop

The slide shows a `FOR` loop that inserts 10 rows into the `order_items` table.

`WHILE` Loop

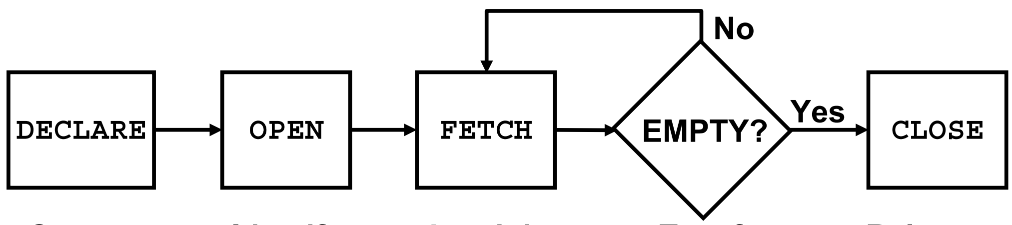
Example:

```
ACCEPT p_price PROMPT 'Enter the price of the item: '
ACCEPT p_itemtot -
 PROMPT 'Enter the maximum total for purchase of item: '
DECLARE
...
v_qty                    NUMBER(8) := 1;
v_running_total          NUMBER(7,2) := 0;
BEGIN
  ...
  WHILE v_running_total < &p_itemtot LOOP
    ...
  v_qty := v_qty + 1;
  v_running_total := v_qty * &p_price;
  END LOOP;
...
```


`WHILE` Loop

In the example in the slide, the quantity increases with each iteration of the loop until the quantity is no longer less than the maximum price allowed for spending on the item.

Controlling Explicit Cursors



- Create a named SQL area.
- Identify the active set.
- Load the current row into variables.
- Test for existing rows.
- Return to *FETCH* if rows are found.
- Release the active set.

Explicit Cursors

Controlling Explicit Cursors Using Four Commands

1. Declare the cursor by naming it and defining the structure of the query to be performed within it.
2. Open the cursor. The `OPEN` statement executes the query and binds any variables that are referenced. Rows identified by the query are called the *active set* and are now available for fetching.
3. Fetch data from the cursor. The `FETCH` statement loads the current row from the cursor into variables. Each fetch causes the cursor to move its pointer to the next row in the active set. Therefore, each fetch accesses a different row returned by the query. In the flow diagram in the slide, each fetch tests the cursor for any existing rows. If rows are found, it loads the current row into variables; otherwise, it closes the cursor.
4. Close the cursor. The `CLOSE` statement releases the active set of rows. It is now possible to reopen the cursor to establish a fresh active set.

Declaring the Cursor

Example:

```
DECLARE
  CURSOR c1 IS
    SELECT employee_id, last_name
    FROM   employees;

  CURSOR c2 IS
    SELECT *
    FROM   departments
    WHERE  department_id = 10;
BEGIN
  ...
```


Explicit Cursor Declaration

Retrieve the employees one by one.

```
DECLARE
  v_empid   employees.employee_id%TYPE;
  v_ename   employees.last_name%TYPE;
  CURSOR c1 IS
    SELECT employee_id, last_name
    FROM   employees;
BEGIN
...
```

Note: You can reference variables in the query, but you must declare them before the `CURSOR` statement.

Opening the Cursor

Syntax:

```
OPEN cursor_name;
```

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

`OPEN` Statement

Open the cursor to execute the query and identify the result set, which consists of all rows that meet the query search criteria. The cursor now points to the first row in the result set.

In the syntax, `cursor_name` is the name of the previously declared cursor.

`OPEN` is an executable statement that performs the following operations:

1. Dynamically allocates memory for a context area that eventually contains crucial processing information
2. Parses the `SELECT` statement
3. Binds the input variables—that is, sets the value for the input variables by obtaining their memory addresses
4. Identifies the result set—that is, the set of rows that satisfy the search criteria. Rows in the result set are not retrieved into variables when the `OPEN` statement is executed. Rather, the `FETCH` statement retrieves the rows.
5. Positions the pointer just before the first row in the active set

Note: If the query returns no rows when the cursor is opened, then PL/SQL does not raise an exception. However, you can test the cursor's status after a fetch.

For cursors declared by using the `FOR UPDATE` clause, the `OPEN` statement also locks those rows.

Fetching Data from the Cursor

Examples:

```
FETCH c1 INTO v_empid, v_ename;
```

```
...
OPEN defined_cursor;
LOOP
  FETCH defined_cursor INTO defined_variables
  EXIT WHEN ...;
  ...
    -- Process the retrieved data
  ...
END;
```


`FETCH` Statement

You use the `FETCH` statement to retrieve the current row values into output variables. After the fetch, you can manipulate the variables by further statements. For each column value returned by the query associated with the cursor, there must be a corresponding variable in the `INTO` list. Also, their data types must be compatible. Retrieve the first 10 employees one by one:

```
DECLARE
  v_empid  employees.employee_id%TYPE;
  v_ename  employees.last_name%TYPE;
  i          NUMBER := 1;
  CURSOR c1 IS
    SELECT employee_id, last_name
    FROM   employees;
BEGIN
  OPEN c1;
  FOR i IN 1..10 LOOP
    FETCH c1 INTO v_empid, v_ename;
    ...
  END LOOP;
END;
```

Closing the Cursor

Syntax:

```
CLOSE   cursor_name;
```

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

CLOSE Statement

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

In the syntax, cursor_name is the name of the previously declared cursor.

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

Note: The CLOSE statement releases the context area. Although it is possible to terminate the PL/SQL block without closing cursors, you should always close any cursor that you declare explicitly in order to free up resources. There is a maximum limit to the number of open cursors per user, which is determined by the OPEN_CURSORS parameter in the database parameter field. By default, the maximum number of OPEN_CURSORS is 50.

```
...
  FOR i IN 1..10 LOOP
    FETCH c1 INTO v_empid, v_ename; ...
  END LOOP;
  CLOSE c1;
END;
```

Explicit Cursor Attributes

Obtain status information about a cursor.

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

Explicit Cursor Attributes

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

Note: Do not reference cursor attributes directly in a SQL statement.

Cursor FOR Loops

Retrieve employees one by one until there are no more left.

Example:

```
DECLARE
  CURSOR c1 IS
    SELECT employee_id, last_name
    FROM   employees;
BEGIN
  FOR emp_record IN c1 LOOP
         -- implicit open and implicit fetch occur
    IF emp_record.employee_id = 134 THEN
      ...
  END LOOP; -- implicit close occurs
END;
```


Cursor FOR Loops

A cursor FOR loop processes rows in an explicit cursor. The cursor is opened, rows are fetched once for each iteration in the loop, and the cursor is closed automatically when all rows have been processed. The loop itself is terminated automatically at the end of the iteration where the last row was fetched.

FOR UPDATE Clause

Retrieve the orders for amounts over $1,000 that were processed today.

Example:

```
DECLARE
  CURSOR c1 IS
    SELECT customer_id, order_id
    FROM   orders
    WHERE  order_date = SYSDATE
      AND  order_total > 1000.00
    ORDER BY customer_id
    FOR UPDATE NOWAIT;
```

ORACLE

FOR UPDATE Clause

If the database server cannot acquire the locks on the rows it needs in a `SELECT FOR UPDATE`, then it waits indefinitely. You can use the `NOWAIT` clause in the `SELECT FOR UPDATE` statement and test for the error code that returns due to failure to acquire the locks in a loop. Therefore, you can retry opening the cursor *n* times before terminating the PL/SQL block.

If you intend to update or delete rows by using the `WHERE CURRENT OF` clause, you must specify a column name in the `FOR UPDATE OF` clause.

If you have a large table, you can achieve better performance by using the `LOCK TABLE` statement to lock all rows in the table. However, when using `LOCK TABLE`, you cannot use the `WHERE CURRENT OF` clause and must use the notation `WHERE` *`column = identifier`*.

WHERE CURRENT OF Clause

Example:

```
DECLARE
  CURSOR c1 IS
    SELECT salary FROM employees
    FOR UPDATE OF salary NOWAIT;
BEGIN
  ...
  FOR emp_record IN c1 LOOP
    UPDATE ...
      WHERE CURRENT OF c1;
    ...
  END LOOP;
  COMMIT;
END;
```


WHERE CURRENT OF Clause

You can update rows based on criteria from a cursor.

Additionally, you can write your `DELETE` or `UPDATE` statement to contain the `WHERE CURRENT OF cursor_name` clause to refer to the latest row processed by the `FETCH` statement. When you use this clause, the cursor you reference must exist and must contain the `FOR UPDATE` clause in the cursor query; otherwise, you get an error. This clause enables you to apply updates and deletes to the currently addressed row without the need to explicitly reference the `ROWID` pseudocolumn.

Trapping Predefined Oracle Server Errors

- Reference the standard name in the exception-handling routine.
- Sample predefined exceptions:
  - `NO_DATA_FOUND`
  - `TOO_MANY_ROWS`
  - `INVALID_CURSOR`
  - `ZERO_DIVIDE`
  - `DUP_VAL_ON_INDEX`

ORACLE

Trapping Predefined Oracle Server Errors

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

Note: PL/SQL declares predefined exceptions in the `STANDARD` package.
It is a good idea to always consider the `NO_DATA_FOUND` and `TOO_MANY_ROWS` exceptions, which are the most common.

Trapping Predefined Oracle Server Errors: Example

Syntax:

```
BEGIN  SELECT ... COMMIT;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    statement1;
    statement2;
  WHEN TOO_MANY_ROWS THEN
    statement1;
  WHEN OTHERS THEN
    statement1;
    statement2;
    statement3;
END;
```


Trapping Predefined Oracle Server Exceptions: Example

In the example in the slide, a message is printed out to the user for each exception. Only one exception is raised and handled at any time.

Non-Predefined Error

Trap for Oracle server error number –2292, which is an integrity constraint violation.

```
DECLARE
  e_products_invalid EXCEPTION;                         1
  PRAGMA EXCEPTION_INIT (
            e_products_invalid, -2292);                 2
  v_message VARCHAR2(50);
BEGIN
. . .                                     3
EXCEPTION
  WHEN e_products_invalid THEN
    :g_message := 'Product ID
            specified is not valid.';
. . .
END;
```


Trapping a Non-Predefined Oracle Server Exception

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

   `*exception*   EXCEPTION;`

   In this syntax, *exception* is the name of the exception.
2. Associate the declared exception with the standard Oracle server error number, using the `PRAGMA EXCEPTION_INIT` statement.
   Syntax

   `PRAGMA EXCEPTION_INIT(exception, error_number);`

   In this syntax:

exception	Is the previously declared exception
error_number	Is a standard Oracle server error number
3. Reference the declared exception within the corresponding exception-handling routine.
   In the slide example: If there is product in stock, halt processing and print a message to the user.

User-Defined Exceptions

Example:

```
[DECLARE]
  e_amount_remaining EXCEPTION;
. . .
BEGIN
. . .
  RAISE e_amount_remaining;
. . .
EXCEPTION
  WHEN e_amount_remaining THEN
    :g_message := 'There is still an amount
                   in stock.';
. . .
END;
```

1 2 3

Trapping User-Defined Exceptions

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

1. Declare the name for the user-defined exception within the declarative section.
   Syntax: *exception* `EXCEPTION;`
   where: *exception* Is the name of the exception
2. Use the `RAISE` statement to raise the exception explicitly within the executable section.
   Syntax: `RAISE` *exception*;
   where: *exception* Is the previously declared exception
3. Reference the declared exception within the corresponding exception-handling routine.

In the slide example: This customer has a business rule that states that a product cannot be removed from its database if there is any inventory left in stock for this product. Because there are no constraints in place to enforce this rule, the developer handles it explicitly in the application. Before performing a `DELETE` on the `PRODUCT_INFORMATION` table, the block queries the `INVENTORIES` table to see whether there is any stock for the product in question. If there is stock, raise an exception.

Note: Use the `RAISE` statement by itself within an exception handler to raise the same exception back to the calling environment.

`RAISE_APPLICATION_ERROR` Procedure

Syntax:

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

- **Enables you to issue user-defined error messages from stored subprograms**
- **Is called from an executing stored subprogram only**

`RAISE_APPLICATION_ERROR` Procedure

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

In the syntax, *`error_number`* is a user-specified number for the exception between –20000 and –20999. The *`message`* is the user-specified message for the exception. It is a character string that is up to 2,048 bytes long.

`TRUE` | `FALSE` is an optional Boolean parameter. If `TRUE`, the error is placed on the stack of previous errors. If `FALSE` (the default), the error replaces all previous errors.

Example:

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

RAISE_APPLICATION_ERROR Procedure

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

ORACLE

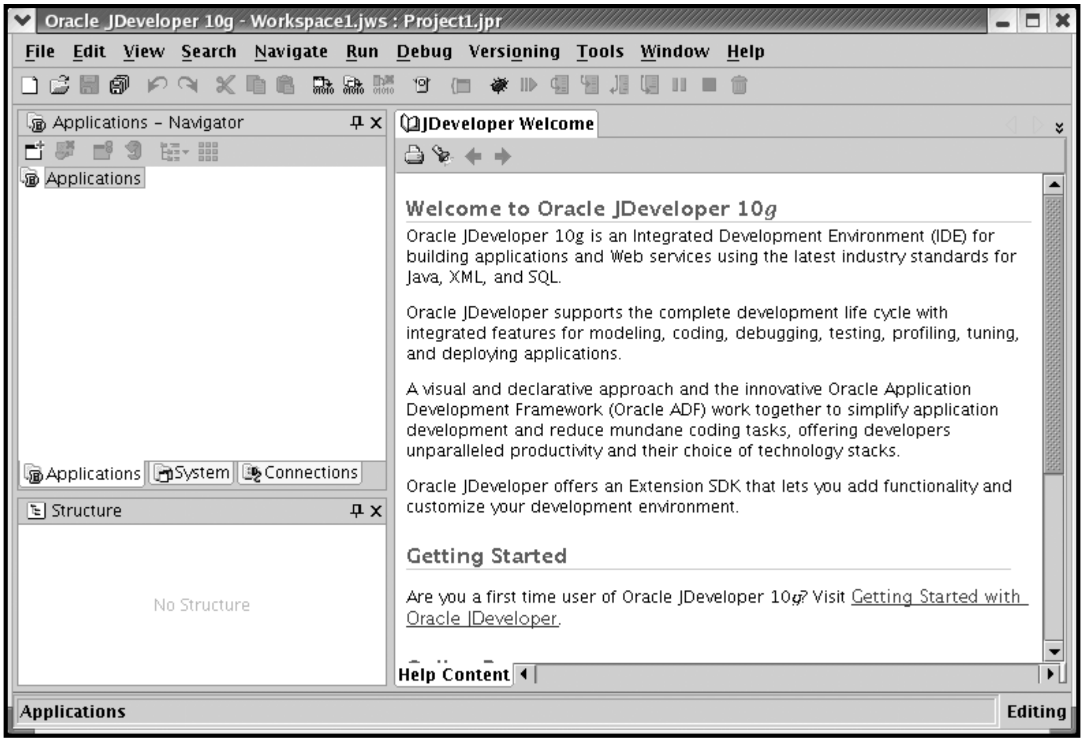
RAISE_APPLICATION_ERROR Procedure: Example

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

E

JDeveloper

JDeveloper



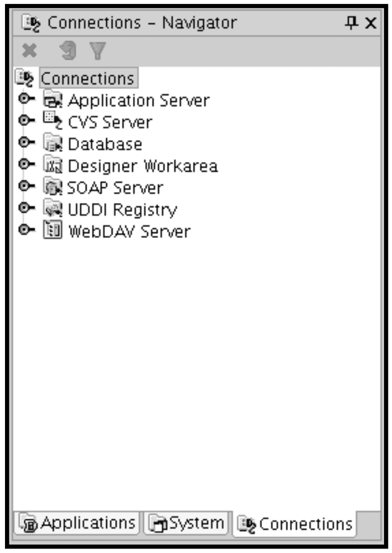
ORACLE

JDeveloper

Oracle JDeveloper 10*g* is an integrated development environment (IDE) for developing and deploying Java applications and Web services. It supports every stage of the software development life cycle (SDLC) from modeling to deploying. It has the features to use the latest industry standards for Java, Extensible Markup Language (XML), and SQL while developing an application.

Oracle JDeveloper 10*g* initiates a new approach to J2EE development with the features that enables visual and declarative development. This innovative approach makes J2EE development simple and efficient.

Connection Navigator

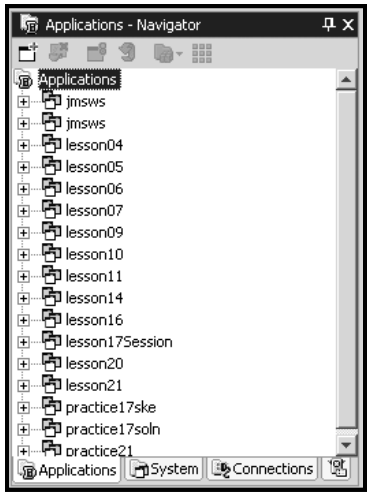


ORACLE

Connection Navigator

Using Oracle JDeveloper 10*g*, you can store the information necessary to connect to a database in an object called "connection." A connection is stored as part of the IDE settings, and can be exported and imported for easy sharing among groups of users. A connection serves several purposes from browsing the database and building applications, all the way through to deployment.

Application Navigator

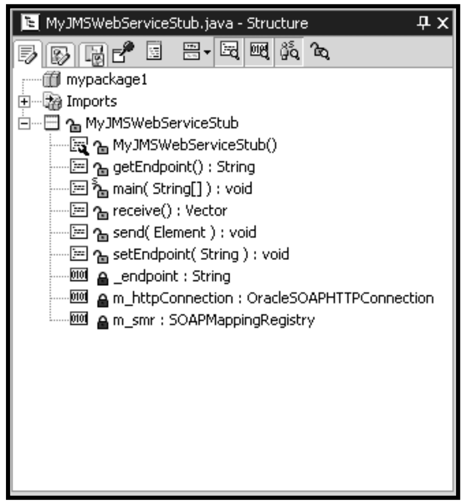


ORACLE

Application Navigator

The Application Navigator gives you a logical view of your application and the data it contains. The Application Navigator provides an infrastructure that the different extensions can plug into and use to organize their data and menus in a consistent, abstract manner. While the Application Navigator can contain individual files (such as Java source files), it is designed to consolidate complex data. Complex data types such as entity objects, UML (Unified Modeling Language) diagrams, Enterprise JavaBeans (EJB), or Web services appear in this navigator as single nodes. The raw files that make up these abstract nodes appear in the Structure window.

Structure Window



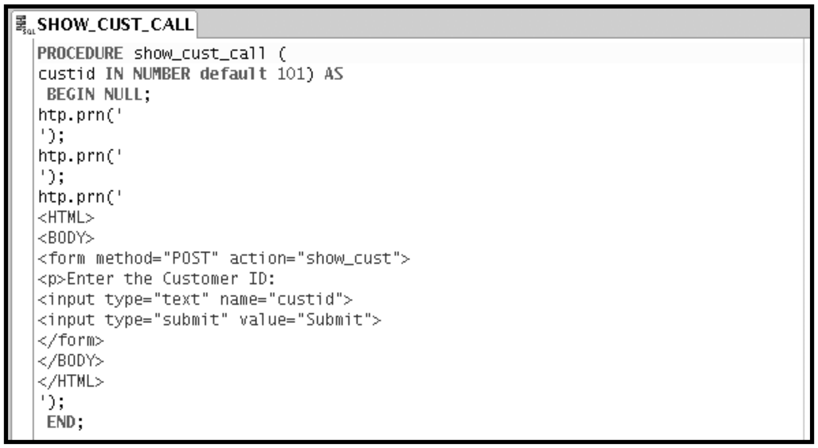
Structure Window

The Structure window offers a structural view of the data in the document currently selected in the active window of those windows that participate in providing structure: the navigators, the editors and viewers, and the Property Inspector.

In the Structure window, you can view the document data in a variety of ways. The structures available for display are based upon document type. For a Java file, you can view code structure, user interface (UI) structure, or UI model data. For an XML file, you can view XML structure, design structure, or UI model data.

The Structure window is dynamic, always tracking the current selection of the active window (unless you freeze the window's contents on a particular view), as is pertinent to the currently active editor. When the current selection is a node in the navigator, the default editor is assumed. To change the view on the structure for the current selection, select a different structure tab.

Editor Window




```
PROCEDURE show_cust_call (
custid IN NUMBER default 101) AS
 BEGIN NULL;
htp.prn('
');
htp.prn('
');
htp.prn('
<HTML>
<BODY>
<form method="POST" action="show_cust">
<p>Enter the Customer ID:
<input type="text" name="custid">
<input type="submit" value="Submit">
</form>
</BODY>
</HTML>
');
 END;
```


Editor Window

You can view all your project files in one single editor window, you can open multiple views of the same file, or you can open multiple views of different files.

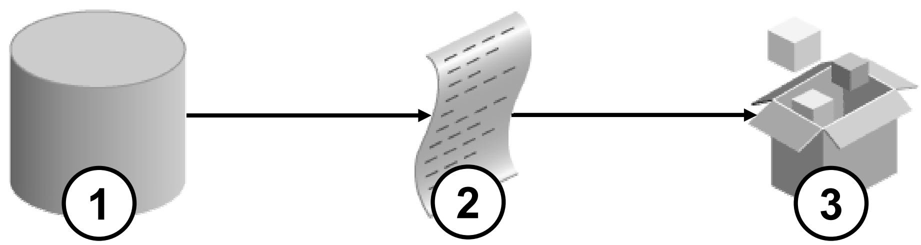
The tabs at the top of the editor window are the document tabs. Selecting a document tab gives that file focus, bringing it to the foreground of the window in the current editor.

The tabs at the bottom of the editor window for a given file are the editor tabs. Selecting an editor tab opens the file in that editor.

Deploying Java Stored Procedures

Before deploying Java stored procedures, perform the following steps:

1. **Create a database connection.**
2. **Create a deployment profile.**
3. **Deploy the objects.**



Deploying Java Stored Procedures

Create a deployment profile for Java stored procedures, then deploy the classes and, optionally, any public static methods in JDeveloper using the settings in the profile.

Deploying to the database uses the information provided in the Deployment Profile Wizard and two Oracle Database utilities:

- `loadjava` loads the Java class containing the stored procedures to an Oracle database.
- `publish` generates the PL/SQL call specific wrappers for the loaded public static methods. Publishing enables the Java methods to be called as PL/SQL functions or procedures.

Publishing Java to PL/SQL

FormatCreditCardNo.java | CCFORMAT

```
public class FormatCreditCardNo
{
  public static final void formatCard(String[] cardno)
  {
  int count=0, space=0;
  String oldcc=cardno[0];
 // System.out.println("Printing the card no initially "+oldcc);
  String[] newcc= {""};
  while (count<16)
  {
  newcc[0]+= oldcc.charAt(count);
  space++;
  if (space ==4)
  {  newcc[0]+=" "; space=0;  }
  count++;
  }
  cardno[0]=newcc [0];
  }
}
```

FormatCreditCardNo.java | CCFORMAT

```
PROCEDURE ccformat (x IN OUT varchar2)
AS LANGUAGE JAVA
NAME 'FormatCreditCardNo.formatCard(java.lang.String[])';
```

ORACLE

Publishing Java to PL/SQL

The slide shows the Java code and how to publish the Java code in a PL/SQL procedure.

Creating Program Units

TEST_JDEV

```
FUNCTION "TEST_JDEV" RETURN VARCHAR2
AS
BEGIN
  RETURN('');
END;
```

Skeleton of the function

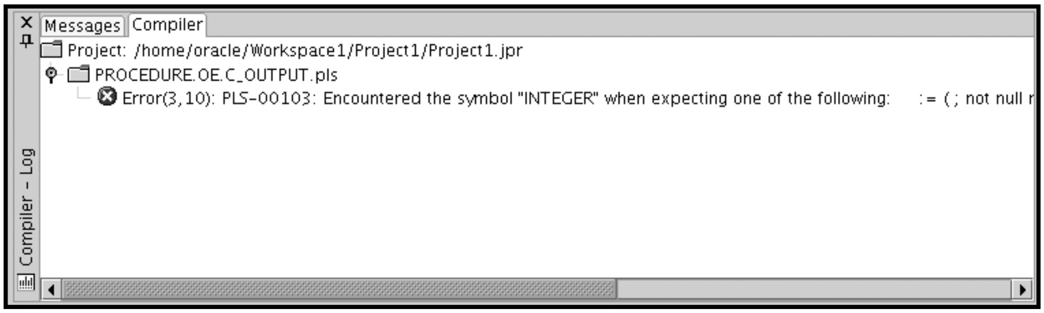
Creating Program Units

To create a PL/SQL program unit:

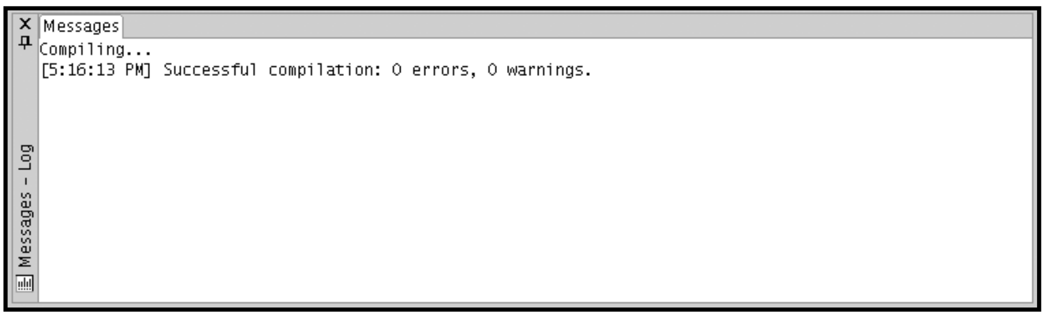
1. Select View > Connection Navigator.
2. Expand Database and select a database connection.
3. In the connection, expand a schema.
4. Right-click a folder corresponding to the object type (Procedures, Packages, and Functions).
5. Choose New PL/SQL object_type. The Create PL/SQL dialog box appears for the function, package, or procedure.
6. Enter a valid name for the function, package, or procedure, and click OK.

A skeleton definition will be created and opened in the Code Editor. You can then edit the subprogram to suit your need.

Compiling



Compilation with errors



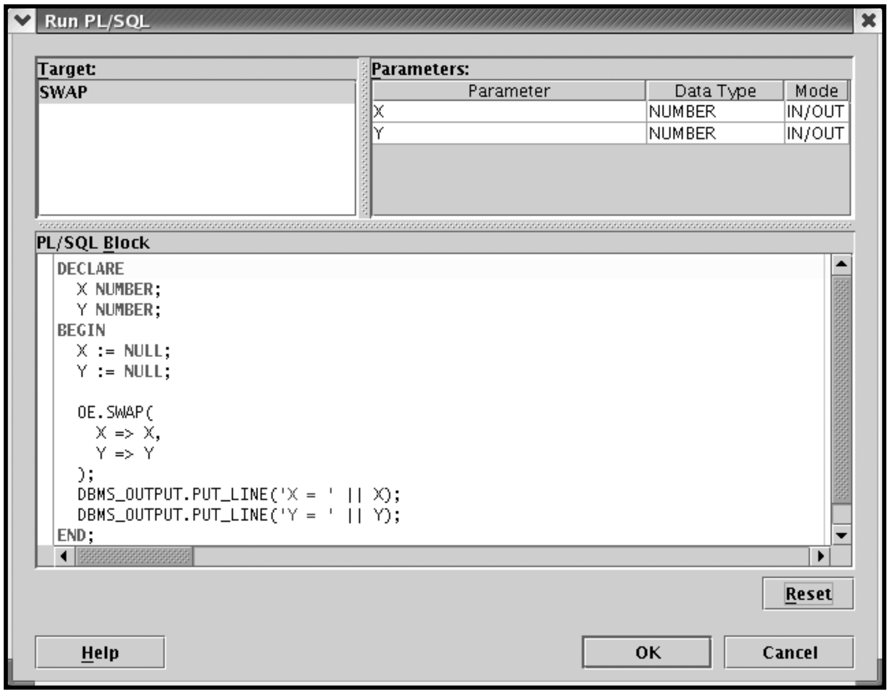
Compilation without errors

ORACLE

Compiling

After editing the skeleton definition, you need to compile the program unit. Right-click the PL/SQL object that you need to compile in the Connection Navigator and then select Compile. Alternatively, you can also press [CTRL] + [SHIFT] + [F9] to compile.

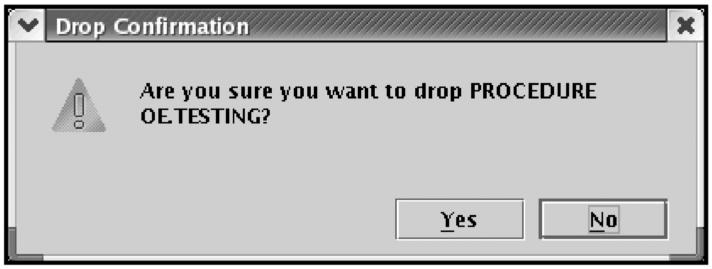
Running a Program Unit



Running a Program Unit

To execute the program unit, right-click the object and click Run. The Run PL/SQL dialog box appears. You may need to change the `NULL` values with reasonable values that are passed into the program unit. After you change the values, click OK. The output will be displayed in the Message-Log window.

Dropping a Program Unit



Dropping a Program Unit

To drop a program unit, right-click the object and select Drop. The Drop Confirmation dialog box appears; click Yes. The object will be dropped from the database.

Debugging PL/SQL Programs

- **JDeveloper support two types of debugging:**
  - **Local**
  - **Remote**
- **You need the following privileges to perform PL/SQL debugging:**
  - **`DEBUG ANY PROCEDURE`**
  - **`DEBUG CONNECT SESSION`**

Debugging PL/SQL Programs

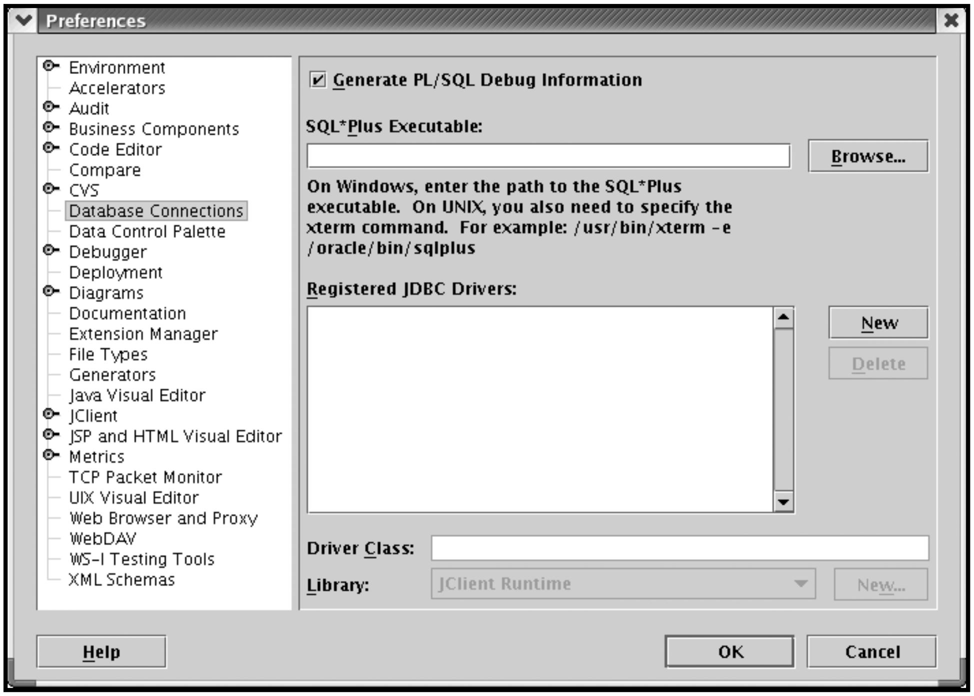
JDeveloper offers both local and remote debugging. A local debugging session is started by setting breakpoints in source files, and then starting the debugger. Remote debugging requires two JDeveloper processes: a `debugger` and a `debuggee`, which may reside on a different platform.

To debug a PL/SQL program, it must be compiled in `INTERPRETED` mode. You cannot debug a PL/SQL program that is compiled in `NATIVE` mode. This mode is set in the database's `init.ora` file.

PL/SQL programs must be compiled with the `DEBUG` option enabled. This option can be enabled using various ways. Using SQL*Plus, execute `ALTER SESSION SET PLSQL_DEBUG = true` to enable the `DEBUG` option. Then you can create or recompile the PL/SQL program you want to debug. Another way of enabling the `DEBUG` option is by using the following command in SQL*Plus:

```
ALTER <procedure, function, package> <name> COMPILE DEBUG;
```

Debugging PL/SQL Programs



Debugging PL/SQL Programs (continued)

Before you start with debugging, make sure that the Generate PL/SQL Debug Information check box is selected. You can access the dialog box by using Tools > Preferences > Database Connections.

Instead of manually testing PL/SQL functions and procedures as you may be accustomed to doing from within SQL*Plus or by running a dummy procedure in the database, JDeveloper enables you to test these objects in an automatic way. With this release of JDeveloper, you can run and debug PL/SQL program units. For example, you can specify parameters being passed or return values from a function giving you more control over what is run and providing you output details about what was tested.

Note: The procedures or functions in the Oracle database can be either stand-alone or within a package.

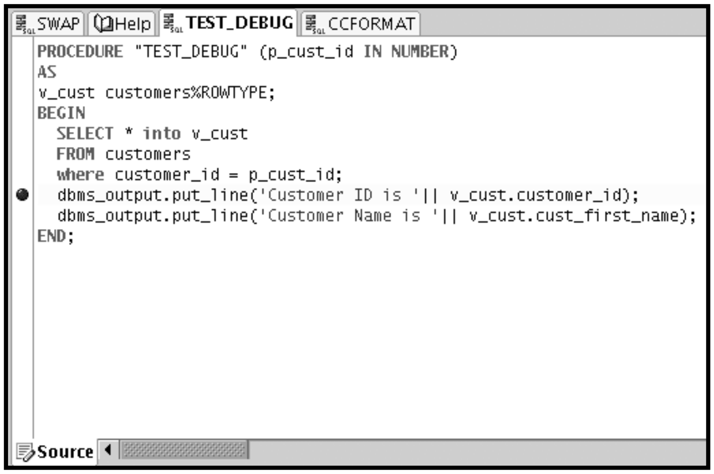
Debugging PL/SQL Programs (continued)

To run or debug functions, procedures, or packages, perform the following steps:

1. Create a database connection by using the Database Wizard.
2. In the Navigator, expand the Database node to display the specific database username and schema name.
3. Expand the Schema node.
4. Expand the appropriate node depending on what you are debugging: Procedure, Function, or Package body.
5. (Optional for debugging only) Select the function, procedure, or package that you want to debug and double-click to open it in the Code Editor.
6. (Optional for debugging only) Set a breakpoint in your PL/SQL code by clicking to the left of the margin.
   Note: The breakpoint must be set on an executable line of code. If the debugger does not stop, the breakpoint may have not been set on an executable line of code (ensure that the breakpoint was verified). Also, verify that the debugging PL/SQL prerequisites were met. In particular, make sure that the PL/SQL program is compiled in `INTERPRETED` mode.
7. Make sure that either the Code Editor or the procedure in the Navigator is currently selected.
8. Click the Debug toolbar button; or, if you want to run without debugging, click the Run toolbar button.
9. The Run PL/SQL dialog box is displayed.
   - Select a target that is the name of the procedure or function that you want to debug. Note that the content in the Parameters and PL/SQL Block boxes change dynamically when the target changes.
     Note: You will have a choice of target only if you choose to run or debug a package that contains more than one program unit.
   - The Parameters box lists the target's arguments (if applicable).
   - The PL/SQL Block box displays code that was custom-generated by JDeveloper for the selected target. Depending on what the function or procedure does, you may need to replace the NULL values with reasonable values so that these are passed into the procedure, function, or package. In some cases, you may need to write additional code to initialize values to be passed as arguments. In this case, you can edit the PL/SQL block text as necessary.
10. Click OK to execute or debug the target.
11. Analyze the output information displayed in the Log window.

In the case of functions, the return value will be displayed. `DBMS_OUTPUT` messages will also be displayed.

Setting Breakpoints

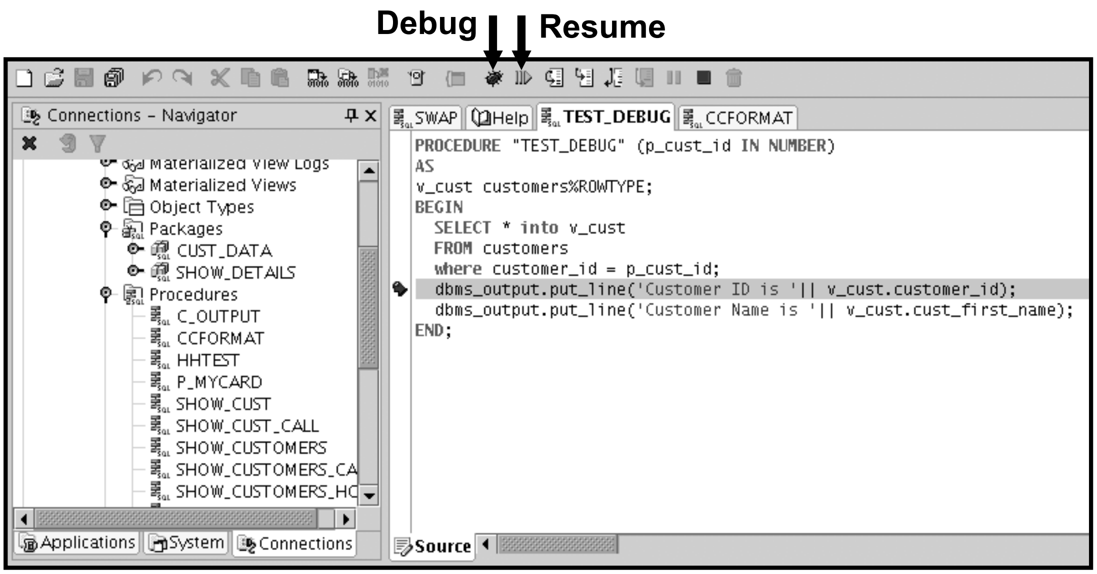


Setting Breakpoints

Breakpoints help you to examine the values of the variables in your program. It is a trigger in a program that, when reached, pauses program execution allowing you to examine the values of some or all of the program variables. By setting breakpoints in potential problem areas of your source code, you can run your program until its execution reaches a location you want to debug. When your program execution encounters a breakpoint, the program pauses, and the debugger displays the line containing the breakpoint in the Code Editor. You can then use the debugger to view the state of your program. Breakpoints are flexible in that they can be set before you begin a program run or at any time while you are debugging.

To set a breakpoint in the Code Editor, click the left margin next to a line of executable code. Breakpoints set on comment lines, blank lines, declaration, and any other nonexecutable lines of code are not verified by the debugger and are treated as invalid.

Stepping Through Code

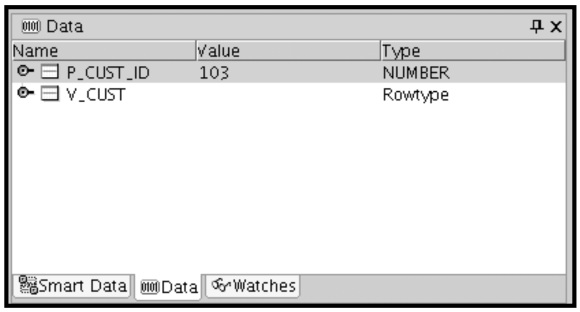


ORACLE

Stepping Through Code

After setting the breakpoint, start the debugger by clicking the Debug icon. The debugger will pause the program execution at the point where the breakpoint is set. At this point, you can check the values of the variables. You can continue with the program execution by clicking the Resume icon. The debugger will then move on to the next breakpoint. After executing all the breakpoints, the debugger will stop the execution of the program and display the results in the Debugging – Log area.

Examining and Modifying Variables



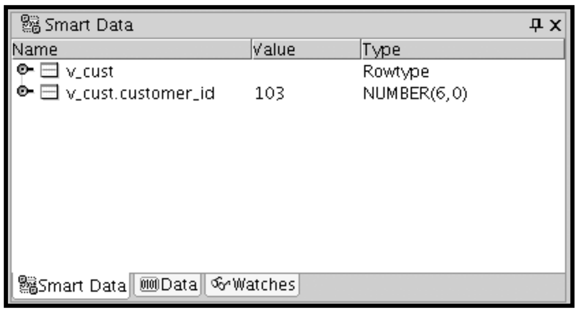
Data window

Examining and Modifying Variables

When the debugger is ON, you can examine and modify the value of the variables using the Data, Smart Data, and Watches windows. You can modify program data values during a debugging session as a way to test hypothetical bug fixes during a program run. If you find that a modification fixes a program error, you can exit the debugging session, fix your program code accordingly, and recompile the program to make the fix permanent.

You use the Data window to display information about variables in your program. The Data window displays the arguments, local variables, and static fields for the current context, which is controlled by the selection in the Stack window. If you move to a new context, the Data window is updated to show the data for the new context. If the current program was compiled without debug information, you will not be able to see the local variables.

Examining and Modifying Variables

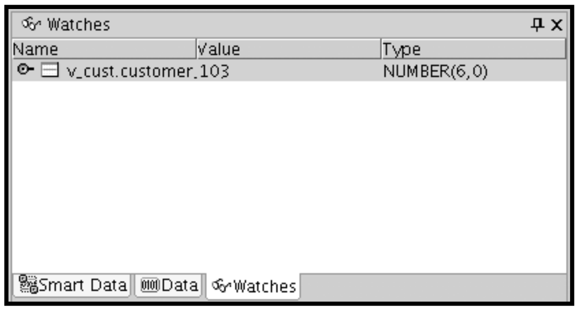


Smart Data window

Examining and Modifying Variables (continued)

Unlike the Data window that displays all the variables in your program, the Smart Data window displays only the data that is relevant to the source code that you are stepping through.

Examining and Modifying Variables

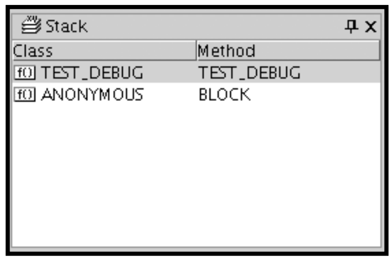


Watches window

Examining and Modifying Variables (continued)

A watch enables you to monitor the changing values of variables or expressions as your program runs. After you enter a watch expression, the Watch window displays the current value of the expression. As your program runs, the value of the watch changes as your program updates the values of the variables in the watch expression.

Examining and Modifying Variables

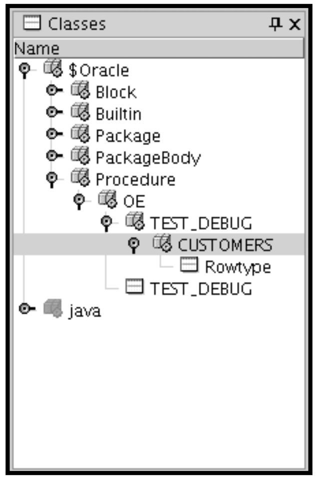


Stack window

Examining and Modifying Variables (continued)

You can activate the Stack window by using View > Debugger > Stack. It displays the call stack for the current thread. When you select a line in the Stack window, the Data window, Watch window, and all other windows are updated to show data for the selected class.

Examining and Modifying Variables



Classes window

Examining and Modifying Variables (continued)

The Classes window displays all the classes that are currently being loaded to execute the program. If used with Oracle Java Virtual Machine (OJVM), it also shows the number of instances of a class and the memory used by those instances.

F

Using SQL Developer

Objectives

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

- **List the key features of Oracle SQL Developer**
- **Install Oracle SQL Developer**
- **Identify menu items of Oracle SQL Developer**
- **Create a database connection**
- **Manage database objects**
- **Use the SQL Worksheet**
- **Execute SQL statements and SQL scripts**
- **Edit and debug PL/SQL statements**
- **Create and save reports**

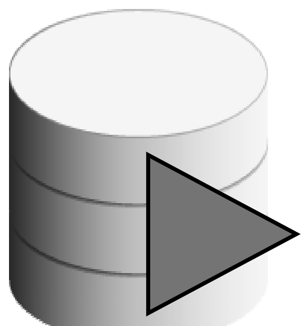
ORACLE

Objectives

This appendix introduces the graphical tool SQL Developer that simplifies your database development tasks. You learn how to use SQL Worksheet to execute SQL statements and SQL scripts. You also learn how to edit and debug PL/SQL.

What Is Oracle SQL Developer?

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



SQL Developer

What Is Oracle SQL Developer?

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

SQL Developer, the visual tool for database development, simplifies the following tasks:

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

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

Key Features

- Developed in Java
- Supports Windows, Linux, and Mac OS X platforms
- Default connectivity by using the JDBC Thin driver
- Does not require an installer
- Connects to any Oracle Database version 9.2.0.1 and later
- Bundled with JRE 1.5

Key Features of SQL Developer

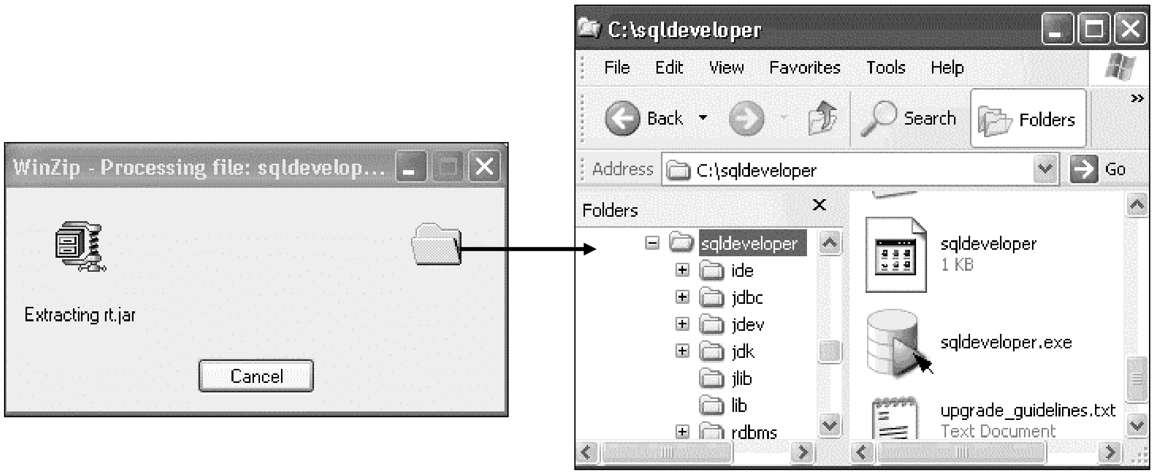
Oracle SQL Developer is developed in Java leveraging the Oracle JDeveloper integrated development environment (IDE). The tool runs on Windows, Linux, and Mac operating system (OS) X platforms. You can install SQL Developer on the Database Server and connect remotely from your desktop, thus avoiding client/server network traffic.

Default connectivity to the database is through the Java Database Connectivity (JDBC) Thin driver, so no Oracle Home is required. SQL Developer does not require an installer and you need to simply unzip the downloaded file.

With SQL Developer, users can connect to Oracle Databases 9.2.0.1 and later, and all Oracle database editions including Express Edition. SQL Developer is bundled with Java Runtime Environment (JRE) 1.5, with an additional `tools.jar` to support Windows clients. Non-Windows clients need only Java Development Kit (JDK) 1.5.

Installing SQL Developer

Download the Oracle SQL Developer kit and unzip it into any directory on your machine.



ORACLE

Installing SQL Developer

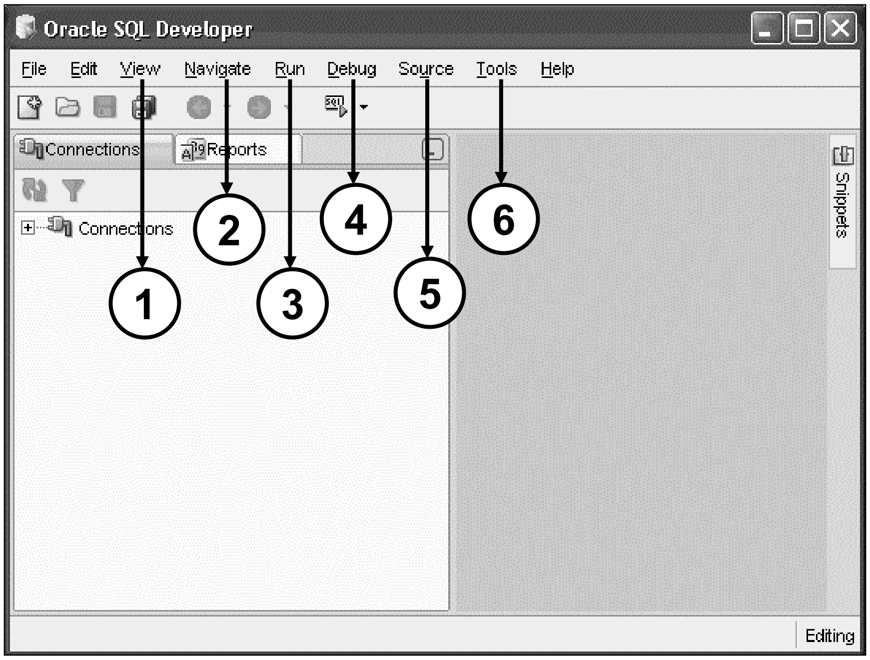
Oracle SQL Developer does not require an installer. To install SQL Developer, you need an unzip tool.

To install SQL Developer, perform the following steps:

1. Create a folder as **`<local drive>:\SQL Developer`**.
2. Download the SQL Developer kit from:
   http://www.oracle.com/technology/software/products/sql/index.html
3. Unzip the downloaded SQL Developer kit into the folder created in step 1.

To start SQL Developer, go to **`<local drive>:\SQL Developer`**, and double-click **`sqldeveloper.exe`**.

Menus for SQL Developer



ORACLE

Menus for SQL Developer

SQL Developer has two main navigation tabs:

- **Connections Navigator:** By using this tab, you can browse database objects and users to which you have access.
- **Reporting Tab:** By using this tab, you can run predefined reports or create and add your own reports.

SQL Developer uses the left side for navigation to find and select objects, and the right side to display information about selected objects. You can customize many aspects of the appearance and behavior of SQL Developer by setting preferences.
The menus at the top contain standard entries, plus entries for features specific to SQL Developer.

1. **View:** Contains options that affect what is displayed in the SQL Developer interface
2. **Navigate:** Contains options for navigating to panes and in the execution of subprograms
3. **Run:** Contains the Run File and Execution Profile options that are relevant when a function or procedure is selected
4. **Debug:** Contains options relevant when a function or procedure is selected
5. **Source:** Contains options for use when editing functions and procedures
6. **Tools:** Invokes SQL Developer tools such as SQL*Plus, Preferences, and SQL Worksheet

Creating a Database Connection

- You must have at least one database connection to use SQL Developer.
- You can create and test connections:
  - For multiple databases
  - For multiple schemas
- SQL Developer automatically imports any connections defined in the `tnsnames.ora` file on your system.
- You can export connections to an XML file.
- Each additional database connection created is listed in the Connections Navigator hierarchy.

ORACLE

Creating a Database Connection

A connection is a SQL Developer object that specifies the necessary information for connecting to a specific database as a specific user of that database. To use SQL Developer, you must have at least one database connection, which may be existing, created, or imported.

You can create and test connections for multiple databases and for multiple schemas.

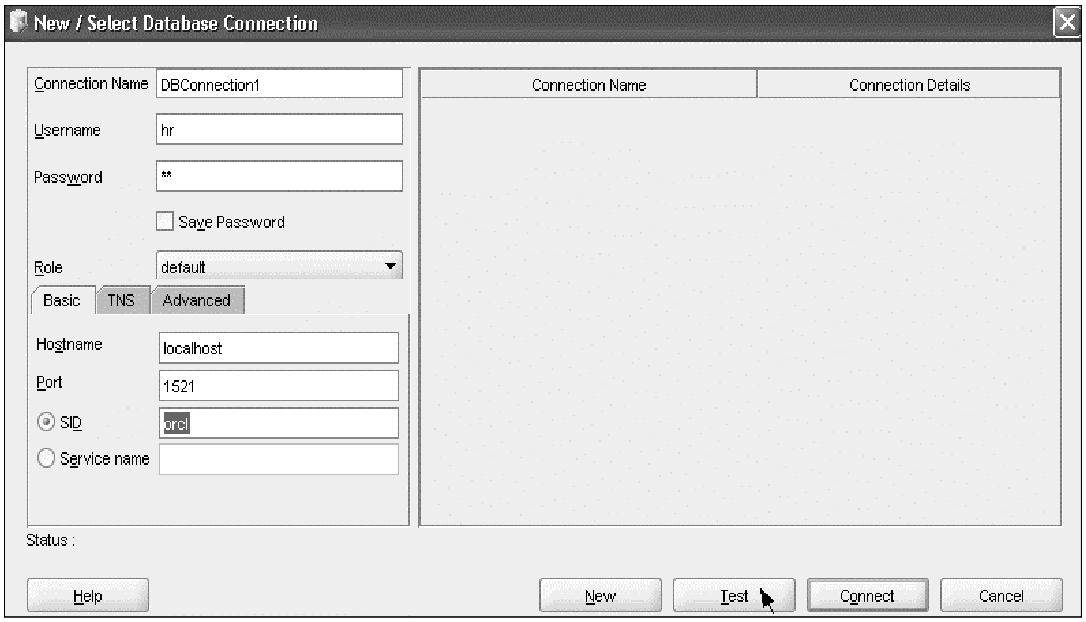
By default, the `tnsnames.ora` file is located in the `$ORACLE_HOME/network/admin` directory. But, it can also be in the directory specified by the `TNS_ADMIN` environment variable or registry value. When you start SQL Developer and display the Database Connections dialog box, SQL Developer automatically imports any connections defined in the `tnsnames.ora` file on your system.

Note: On Windows systems, if the `tnsnames.ora` file exists but its connections are not being used by SQL Developer, define `TNS_ADMIN` as a system environment variable.

You can export connections to an XML file so that you can reuse it later.

You can create additional connections as different users to the same database or to connect to the different databases.

Creating a Database Connection



ORACLE

Creating a Database Connection (continued)

To create a database connection, perform the following steps:

1. Double-click **<*your_path*>\sqldeveloper\sqldeveloper.exe**.
2. On the Connections tabbed page, right-click **Connections** and select **New Database Connection**.
3. Enter the connection name, username, password, hostname, and SID for the database you want to connect.
4. Click **Test** to make sure that the connection has been set correctly.
5. Click **Connect**.

On the basic tabbed page, at the bottom, enter the following options:

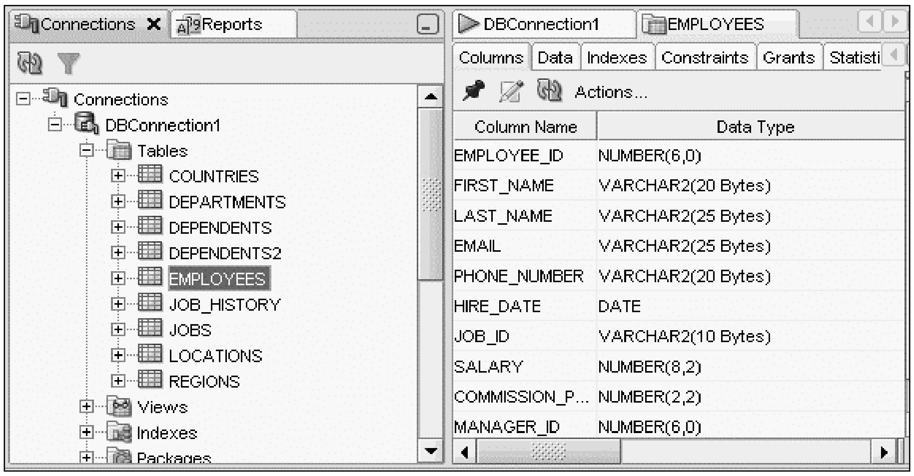
- **Hostname:** Host system for the Oracle database
- **Port:** Listener port
- **SID:** Database name
- **Service Name:** Network service name for a remote database connection

If you select the Save Password check box, the password is saved to an XML file. So, after you close the SQL Developer connection and open it again, you will not be prompted for the password.

Browsing Database Objects

Use the Database Navigator to:

- Browse through many objects in a database schema
- Review the definitions of objects at a glance



Browsing Database Objects

After you have created a database connection, you can use the Database Navigator to browse through many objects in a database schema including Tables, Views, Indexes, Packages, Procedures, Triggers, Types, and so on.

SQL Developer uses the left side for navigation to find and select objects, and the right side to display information about the selected objects. You can customize many aspects of the appearance of SQL Developer by setting preferences.

You can see the definition of the objects broken into tabs of information that is pulled out of the data dictionary. For example, if you select a table in the Navigator, the details about columns, constraints, grants, statistics, triggers, and so on are displayed in an easy-to-read tabbed page.

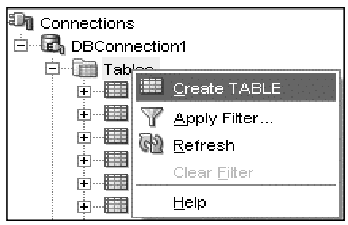
If you want to see the definition of the `EMPLOYEES` table as shown in the slide, perform the following steps:

1. Expand the Connections node in the Connections Navigator.
2. Expand **Tables**.
3. Double-click **EMPLOYEES**.

Using the Data tab, you can enter new rows, update data, and commit these changes to the database.

Creating a Schema Object

- **SQL Developer supports the creation of any schema object by:**
  - **Executing a SQL statement in the SQL Worksheet**
  - **Using the context menu**
- **Edit the objects using an edit dialog box or one of the many context-sensitive menus.**
- **View the DDL for adjustments such as creating a new object or editing an existing schema object.**



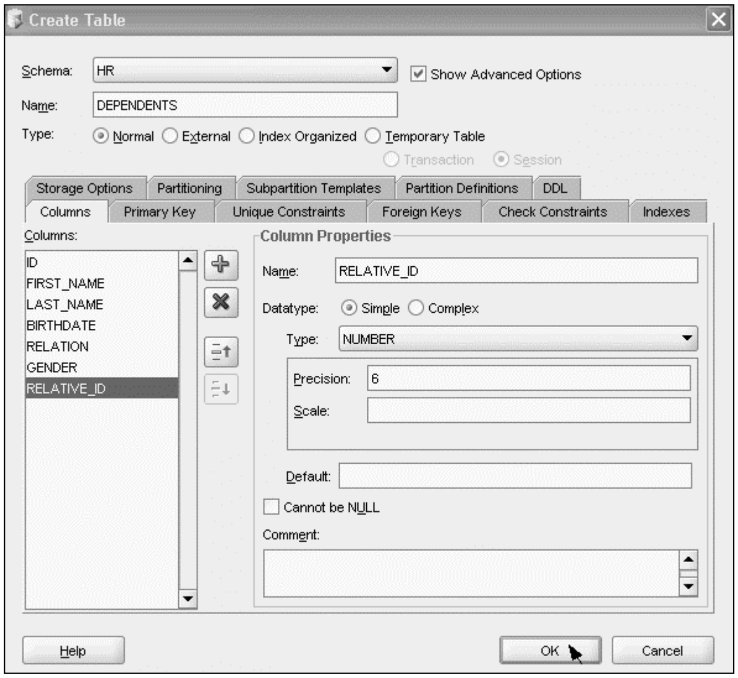
Creating a Schema Object

SQL Developer supports the creation of any schema object by executing a SQL statement in the SQL Worksheet. Alternatively, you can create objects using the context menus. After the objects are created, you can edit the objects using an edit dialog box or one of the many context-sensitive menus.

As new objects are created or existing objects are edited, the data definition language (DDL) for those adjustments is available for review. An Export DDL option is available if you want to create the full DDL for one or more objects in the schema.

The slide shows creating a table using the context menu. To open a dialog box for creating a new table, right-click **Tables** and select **Create TABLE**. The dialog boxes for creating and editing database objects have multiple tabs, each reflecting a logical grouping of properties for that type of object.

Creating a New Table: Example



Creating a New Table: Example

In the Create Table dialog box, if you do not select the **Show Advanced Options** check box, you can create a table quickly by specifying columns and some frequently used features.

If you select the **Show Advanced Options** check box, the Create Table dialog box changes to one with multiple tabs, in which you can specify an extended set of features while creating the table.

The example in the slide shows creating the `DEPENDENTS` table by selecting the **Show Advanced Options** check box.

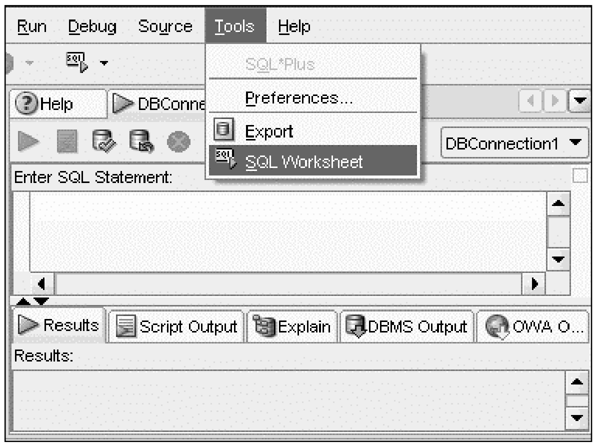
To create a new table, perform the following steps:

1. In the Connections Navigator, right-click **Tables**.
2. Select **Create TABLE**.
3. In the Create Table dialog box, select **Show Advanced Options**.
4. Specify column information.
5. Click **OK**.

Although it is not required, you should also specify a primary key using the Primary Key tab in the dialog box. Sometimes, you may want to edit the table that you have created. To edit a table, right-click the table in the Connections Navigator, and select **Edit**.

Using the SQL Worksheet

- **Use the SQL Worksheet to enter and execute SQL, PL/SQL, and SQL *Plus statements.**
- **Specify any actions that can be processed by the database connection associated with the worksheet.**



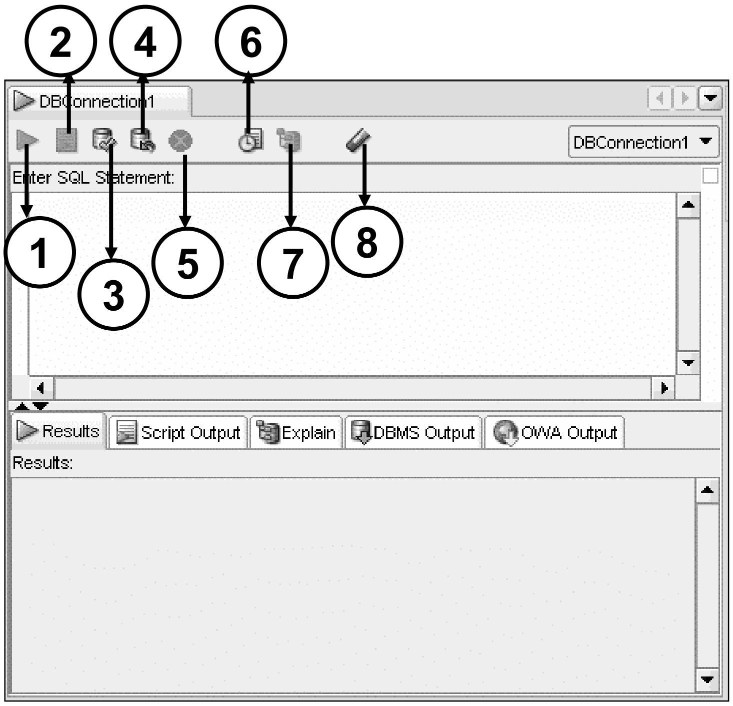
Using the SQL Worksheet

When you connect to a database, a SQL Worksheet window for that connection is automatically opened. You can use the SQL Worksheet to enter and execute SQL, PL/SQL, and SQL*Plus statements. The SQL Worksheet supports SQL*Plus statements to a certain extent. SQL*Plus statements that are not supported by the SQL Worksheet are ignored and not passed to the database.

You can specify any actions that can be processed by the database connection associated with the worksheet, such as:
- Creating a table
- Inserting data
- Creating and editing a trigger
- Selecting data from a table
- Saving the selected data to a file

You can display a SQL Worksheet by using any of the following two options:
- Select **Tools > SQL Worksheet**
- Click the **Open SQL Worksheet** icon.

Using the SQL Worksheet



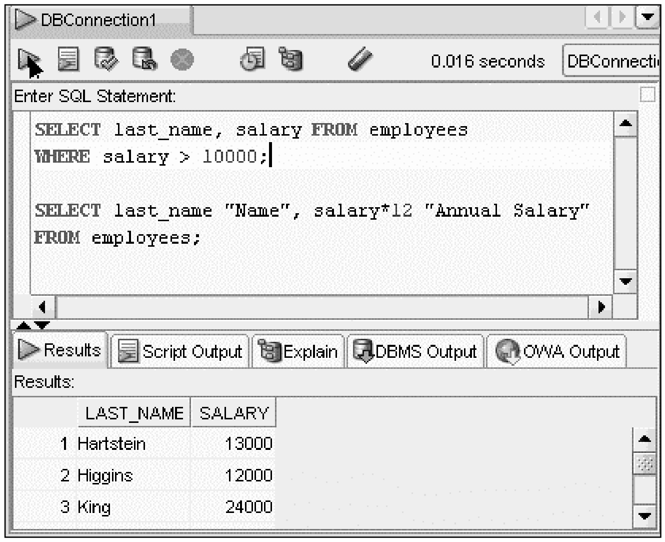
Using the SQL Worksheet (continued)

You may want to use shortcut keys or icons to perform certain tasks such as executing a SQL statement, running a script, and viewing the history of SQL statements that you have executed. You can use the SQL Worksheet toolbar that contains icons to perform the following tasks:

1. **Execute Statement:** Executes the statement at the cursor in the Enter SQL Statement box. You can use bind variables in the SQL statements but not substitution variables.
2. **Run Script:** Executes all statements in the Enter SQL Statement box using the Script Runner. You can use substitution variables in the SQL statements but not bind variables.
3. **Commit:** Writes any changes to the database, and ends the transaction
4. **Rollback:** Discards any changes to the database, without writing them to the database, and ends the transaction
5. **Cancel:** Stops the execution of any statements currently being executed
6. **SQL History:** Displays a dialog box with information about SQL statements that you have executed
7. **Execute Explain Plan:** Generates the execution plan, which you can see by clicking the Explain tab
8. **Clear:** Erases the statement or statements in the Enter SQL Statement box

Executing SQL Statements

Use the Enter SQL Statement box to enter single or multiple SQL statements.



ORACLE

Executing SQL Statements

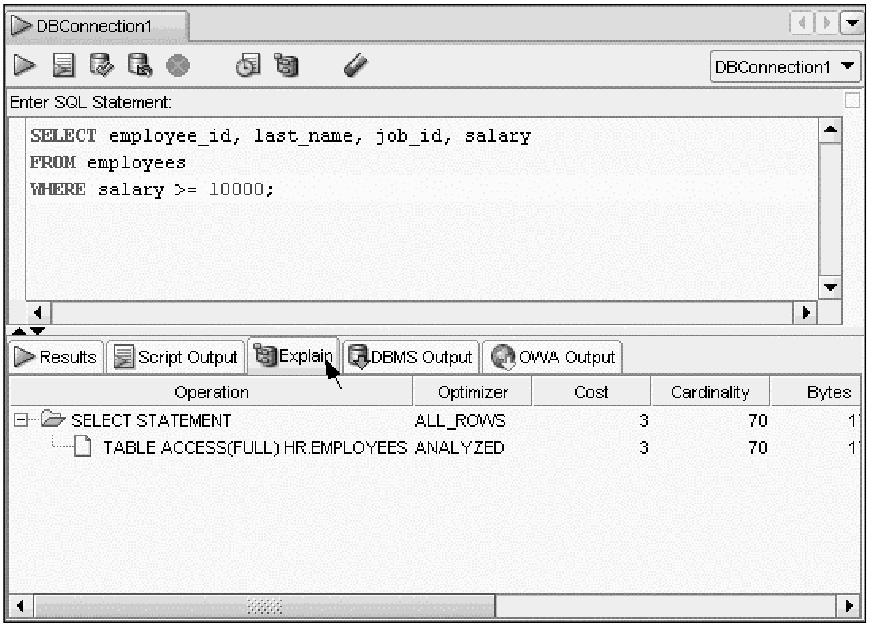
In the SQL Worksheet, you can use the Enter SQL Statement box to type a single or multiple SQL statements. For a single statement, the semicolon at the end is optional.

When you type in the statement, the SQL keywords are automatically highlighted. To execute a SQL statement, ensure that your cursor is within the statement and click the **Execute Statement** icon. Alternatively, you can press the **F9** key.

To execute multiple SQL statements and see the results, click the **Run Script** icon. Alternatively, you can press the **F5** key.

In the example in the slide, because there are multiple SQL statements, the first statement is terminated with a semicolon. The cursor is in the first statement and so when the statement is executed, results corresponding to the first statement are displayed in the Results box.

Viewing the Execution Plan



Viewing the Execution Plan

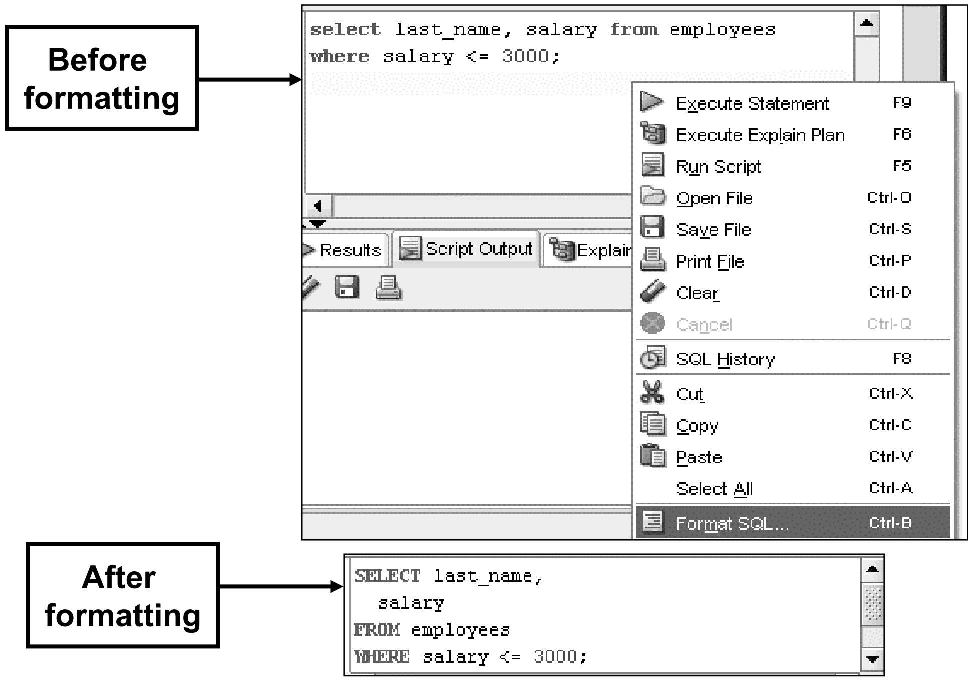
You can execute a SQL script, and view the execution plan. To execute a SQL script file, perform the following steps:

1. Right-click in the Enter SQL Statement box, and select **Open File** from the drop-down menu.
2. In the Open dialog box, double-click the **`.sql`** file.
3. Click the **Run Script** icon.

When you double-click the `.sql` file, the SQL statements are loaded into the Enter SQL Statement box. You can execute the script or each line individually. The results are displayed in the Script Output area.

The example in the slide shows the execution plan. The Execute Explain Plan icon generates the execution plan. An execution plan is the sequence of operations that will be performed to execute the statement. You can see the execution plan by clicking the **Explain** tab.

Formatting the SQL Code



Formatting the SQL Code

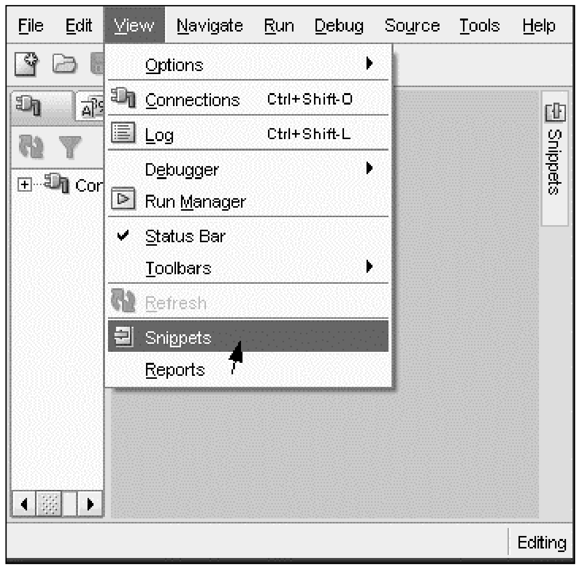
You may want to beautify the indentation, spacing, capitalization, and line separation of the SQL code. SQL Developer has the feature of formatting the SQL code.

To format the SQL code, right-click in the statement area, and select **Format SQL**.

In the example in the slide, before formatting, the SQL code has the keywords not capitalized and the statement not properly indented. After formatting, the SQL code is beautified with the keywords capitalized and the statement properly indented.

Using Snippets

Snippets are code fragments that may be just syntax or examples.



Using Snippets

You may want to use certain code fragments when you are using the SQL Worksheet or creating or editing a PL/SQL function or procedure. SQL Developer has the feature called Snippets. Snippets are code fragments, such as SQL functions, Optimizer hints, and miscellaneous PL/SQL programming techniques. You can drag snippets into the Editor window.

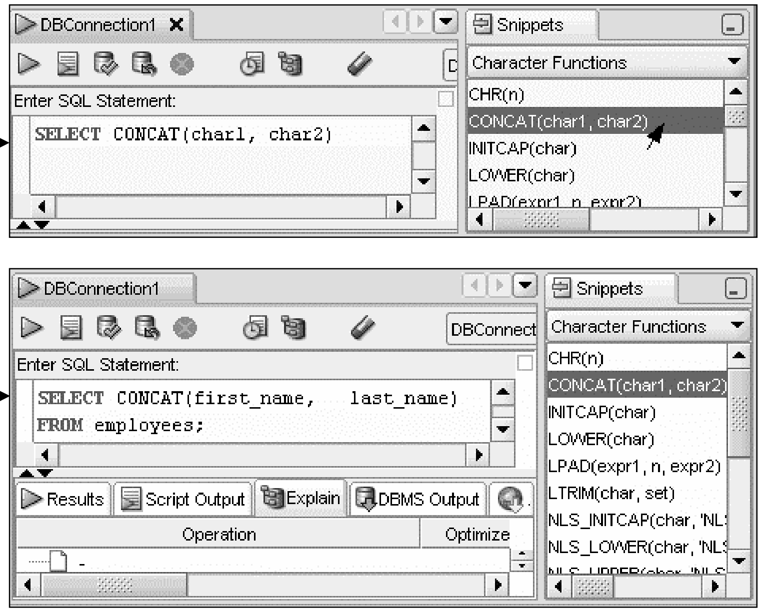
To display Snippets, select **View > Snippets**.

The Snippets window is displayed on the right side. You can use the drop-down list to select a group. A Snippets button is placed in the right window margin, so that you can display the Snippets window if it becomes hidden.

Using Snippets: Example

Inserting a snippet

Editing the snippet



Using Snippets: Example

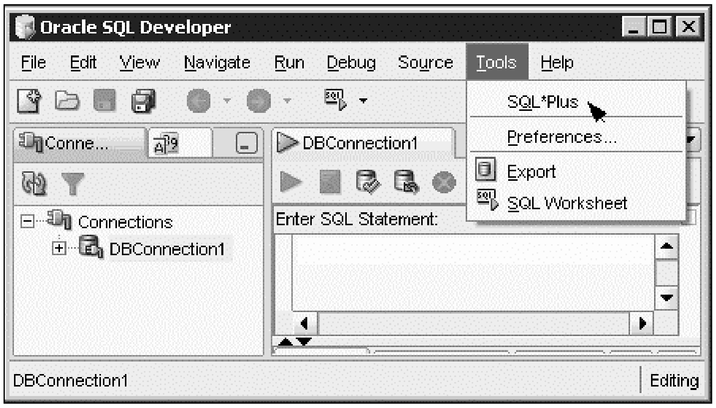
To insert a Snippet into your code in a SQL Worksheet or in a PL/SQL function or procedure, drag the snippet from the Snippets window into the desired place in your code. Then you can edit the syntax so that the SQL function is valid in the current context. To see a brief description of a SQL function in a tool tip, place the cursor over the function name.

The example in the slide shows that `CONCAT(char1, char2)` is dragged from the Character Functions group in the Snippets window. Then the `CONCAT` function syntax is edited and the rest of the statement is added such as in the following:

```
SELECT CONCAT(first_name, last_name)
FROM employees;
```

Using SQL*Plus

- **The SQL Worksheet does not support all SQL*Plus statements.**
- **You can invoke the SQL*Plus command-line interface from SQL Developer.**



Using SQL*Plus

The SQL Worksheet supports some SQL*Plus statements. SQL*Plus statements must be interpreted by the SQL Worksheet before being passed to the database; any SQL*Plus statements that are not supported by the SQL Worksheet are ignored and not passed to the database. For example, some of the SQL*Plus statements that are not supported by SQL Worksheet are:

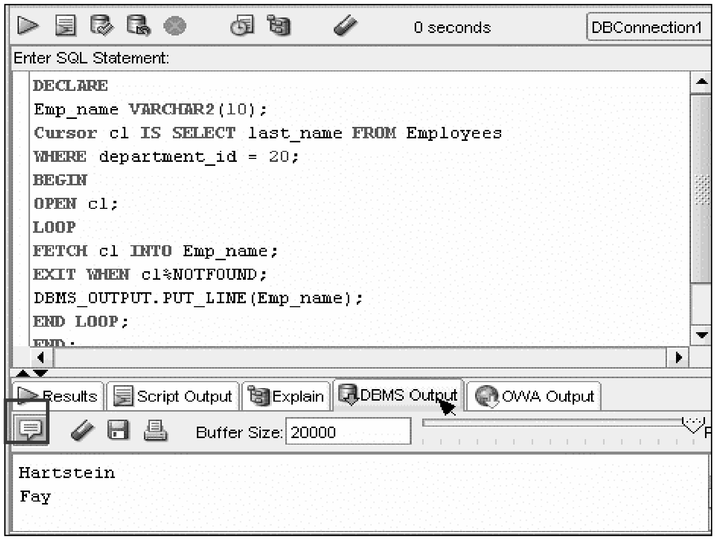
- `append`
- `archive`
- `attribute`
- `break`

For the complete list of SQL*Plus statements that are supported, and not supported by SQL Worksheet, refer to SQL Developer online Help.

To display the SQL*Plus command window, from the Tools menu, select **SQL*Plus**. To use this feature, the system on which you are using SQL Developer must have an Oracle Home directory or folder, with a SQL*Plus executable under that location. If the location of the SQL*Plus executable is not already stored in your SQL Developer preferences, you are asked to specify its location.

Creating an Anonymous Block

Create an anonymous block and display the output of the `DBMS_OUTPUT` package statements.



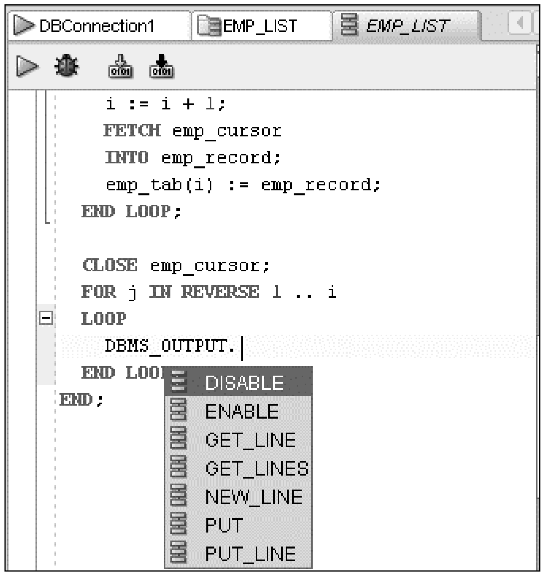
Creating an Anonymous Block

You can create an anonymous block and display the output of the `DBMS_OUTPUT` package statements. To create an anonymous block and view the results, perform the following steps:

1. Enter the PL/SQL code in the Enter SQL Statement box.
2. Click the **DBMS Output** pane. Then click the **Enable DBMS Output** icon to set the server output ON.
3. Click the **Execute Statement** icon above the Enter SQL Statement box. Then click the **DBMS Output** pane to see the results.

Editing the PL/SQL Code

Use the full-featured editor for PL/SQL program units.



ORACLE

Editing the PL/SQL Code

You may want to make changes to your PL/SQL code. SQL Developer includes a full-featured editor for PL/SQL program units. It includes customizable PL/SQL syntax highlighting in addition to common editor functions such as:

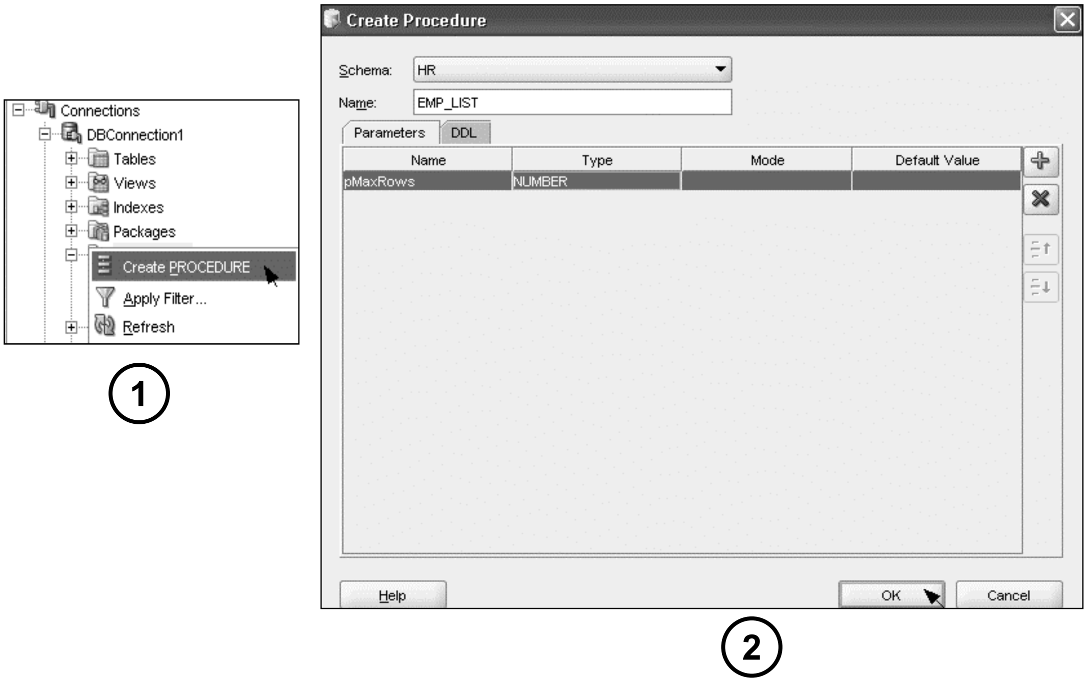
- Bookmarks
- Code Completion
- Code Folding
- Search and Replace

To edit the PL/SQL code, click the object name in the Connections Navigator, and then click the **Edit** icon. Optionally, double-click the object name to invoke the Object Definition page with its tabs and the Edit page. You can update only if you are on the Edit tabbed page.

The Code Insight feature is shown in the slide. For example, if you type `DBMS_OUTPUT.` and then press [Ctrl] + [Spacebar], you can select from a list of members of that package. Note that, by default, Code Insight is invoked automatically if you pause after typing a period (".") for more than one second.

When using the Code Editor to edit PL/SQL code, you can Compile or Compile for Debug.

Creating a PL/SQL Procedure



Creating a PL/SQL Procedure

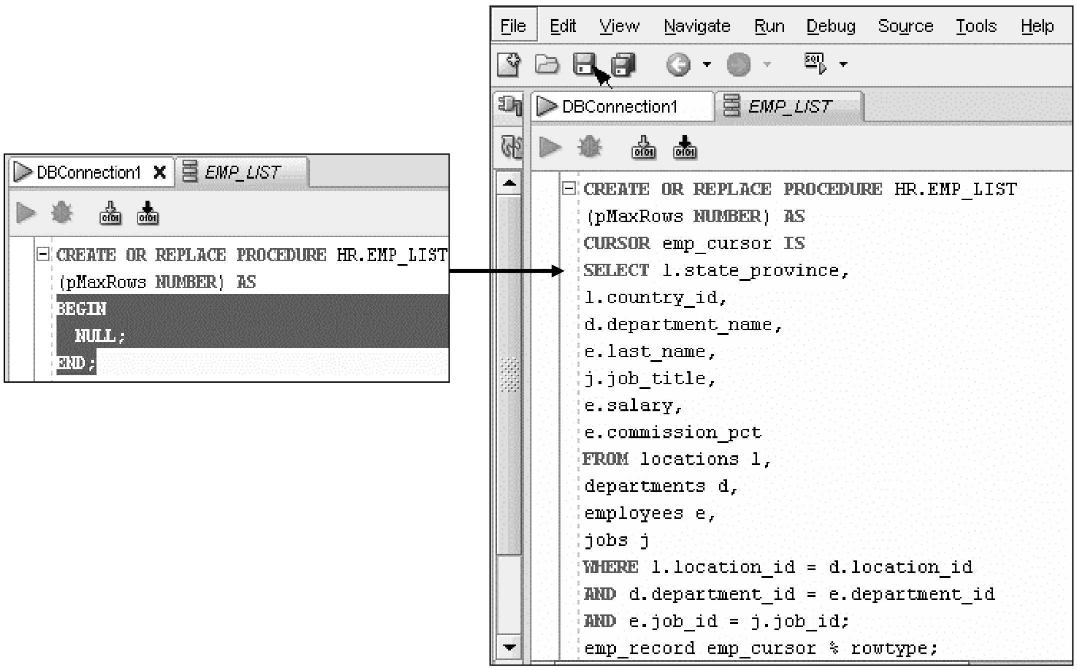
Using SQL Developer, you can create PL/SQL functions, procedures, and packages. To create a PL/SQL procedure, perform the following steps:

1. Right-click the Procedures node in the Connections Navigator to invoke the Context menu, and select **Create Procedure**.
2. In the Create Procedure dialog box, specify the procedure information and click **OK**.

Note: Ensure that you press Enter before you click OK.

In the example in the slide, the `EMP_LIST` procedure is created. The default values for parameter name and parameter type are replaced with `pMaxRows` and `NUMBER`, respectively.

Compiling a PL/SQL Procedure



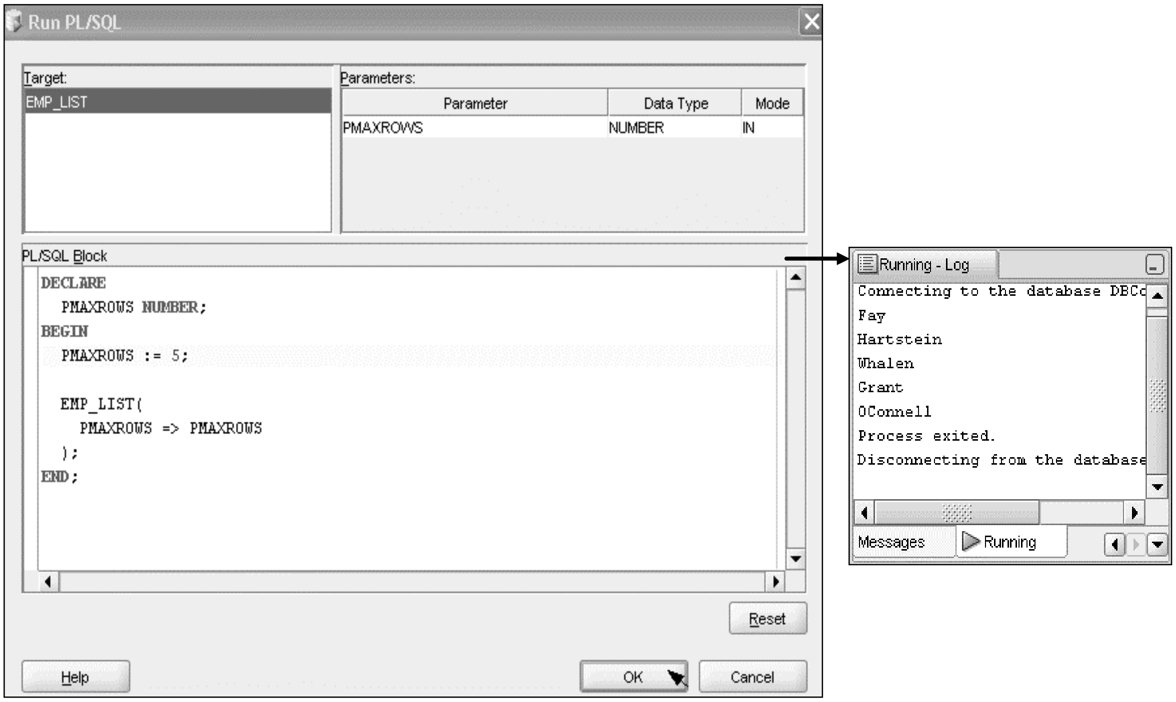
Compiling a PL/SQL Procedure

After you specify the parameter information in the Create Procedure dialog box and click OK, you see the Procedure tab added in the right window. You can then replace the Anonymous block with your PL/SQL code.

To compile the PL/SQL subprogram, click the Save button in the toolbar. If you expand Procedures in the Connections Navigator, you can see that the Procedure node is added.

When an invalid PL/SQL subprogram is detected by SQL Developer, the status is indicated with a red X over the icon for the subprogram in the Connections Navigator. Compilation errors are shown in the log window. You can navigate to the line reported in the error by simply double-clicking the error. SQL Developer also displays errors and hints in the right-hand gutter. If you place the cursor over each of the red bars in the gutter, the error message appears. For example, if the error messages indicate that there is a formatting error, modify the code accordingly and click the Compile icon.

Running a PL/SQL Procedure



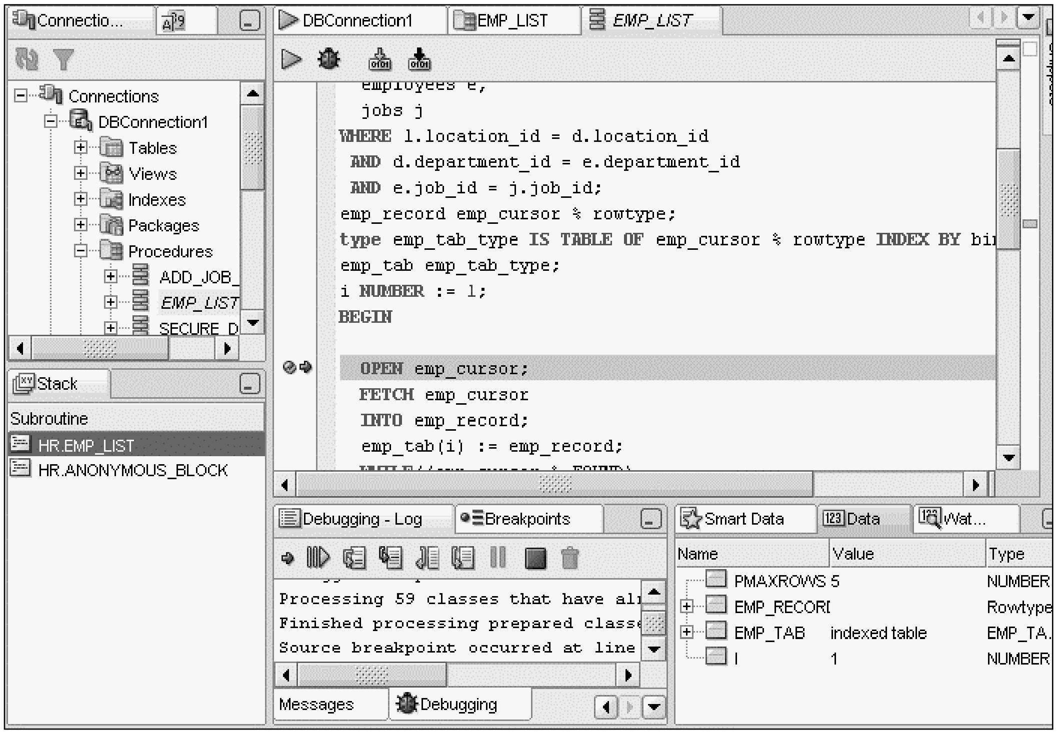
Running a PL/SQL Procedure

After you have created and compiled a PL/SQL procedure, you can run it using SQL Developer. To run a PL/SQL procedure, right-click the procedure name in the left navigator and select Run. Optionally, you can use the Run button in the right window. This invokes the Run PL/SQL dialog box. The Run PL/SQL dialog box enables you to select the target procedure or function to run and displays a list of parameters for the selected target.

You can use the PL/SQL block area to populate parameters to be passed to the program unit and to handle complex return types. After you make the necessary changes in the Run PL/SQL dialog box, click **OK**. You see the expected results in the Running-Log window.

In the example in the slide, `PMAXROWS := NULL;` is changed to `PMAXROWS := 5;`. The results of the five rows returned are displayed in the Running-Log window.

Debugging PL/SQL



Debugging PL/SQL

You may want to debug a PL/SQL function, procedure, or package. SQL Developer provides full support for PL/SQL debugging. To debug a function or procedure, perform the following steps:

1. Click the object name in the Connections Navigator.
2. Right-click the object and select **Compile for debug**.
3. Click the **Edit** icon. Then click the **Debug** icon above its source listing.

If the toggle numbers before each line of code is not yet displayed, right-click in the Code Editor margin and select Toggle Line Numbers.

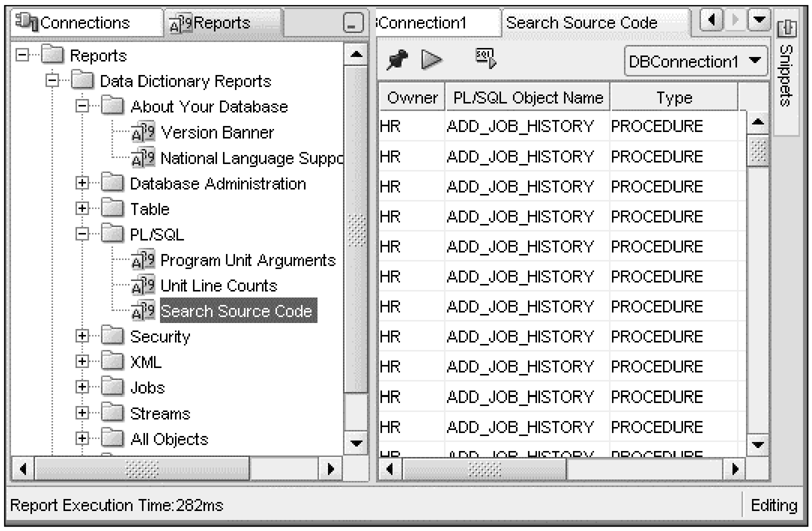
The PL/SQL debugger supplies many commands to control program execution including Step Into, Step Over, Step Out, Run to Cursor, and so on. While the debugger is paused, you can examine and modify the values of variables from the Smart Data, Watches, or Inspector windows.

The Breakpoints window lists the defined breakpoints. You can use this window to add new breakpoints, or customize the behavior of existing breakpoints.

Note: For PL/SQL debugging, you need the `debug any procedure` and `debug connect session` privileges.

Database Reporting

SQL Developer provides a number of predefined reports about the database and its objects.



Database Reporting

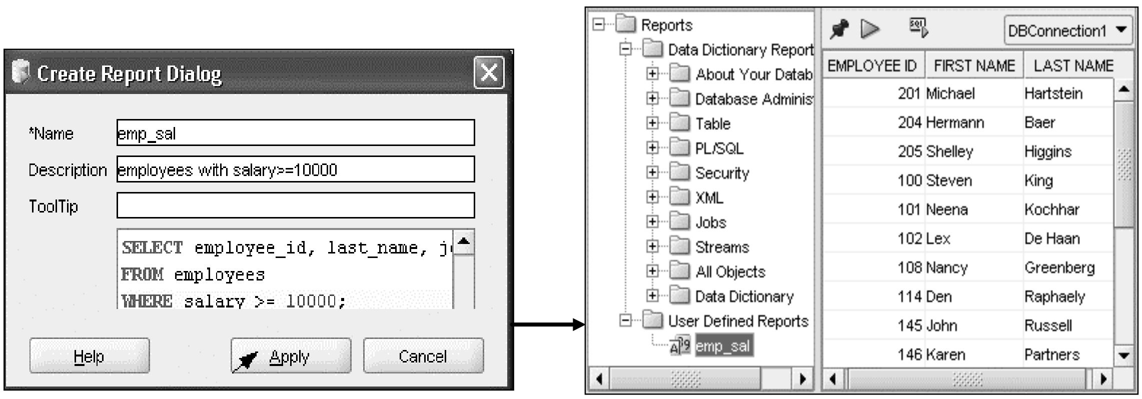
SQL Developer provides many reports about the database and its objects. These reports can be grouped into the following categories:

- About Your Database reports
- Database Administration reports
- Table reports
- PL/SQL reports
- Security reports
- XML reports
- Jobs reports
- Streams reports
- All Objects reports
- Data Dictionary reports
- User Defined reports

To display reports, click the Reports tab on the left side of the window. Individual reports are displayed in tabbed panes on the right side of the window; and for each report, you can select (using a drop-down list) the database connection for which to display the report. For reports about objects, the objects shown are only those visible to the database user associated with the selected database connection, and the rows are usually ordered by Owner.

Creating a User-Defined Report

Create and save user-defined reports for repeated use.



ORACLE

Creating a User-Defined Report

User-defined reports are any reports that are created by SQL Developer users. To create a user-defined report, perform the following steps:

1. Right-click the **User Defined Reports** node under Reports, and select **Add Report**.
2. In the Create Report Dialog box, specify the report name and the SQL query to retrieve information for the report. Then click **Apply**.

In the example in the slide, the report name is specified as `emp_sal`. An optional description is provided indicating that the report contains details about employees with salary equal to or greater than 10,000. The complete SQL statement for retrieving the information to be displayed in the user-defined report is specified in the SQL box. You can also include an optional tool tip to be displayed when the cursor stays briefly over the report name in the Reports Navigator display.

You can organize user-defined reports in folders, and you can create a hierarchy of folders and subfolders. To create a folder for user-defined reports, right-click the User Defined node or any folder name under that node and select Add Folder.

Information about user-defined reports, including any folders for these reports, is stored in a file named `UserReports.xml` under the directory for user-specific information.

Summary

In this appendix, you should have learned how to use SQL Developer to do the following:
- **Browse, create, and edit database objects**
- **Execute SQL statements and scripts in the SQL Worksheet**
- **Edit and debug PL/SQL statements**
- **Create and save custom reports**

Summary

SQL Developer is a free graphical tool to simplify database development tasks. Using SQL Developer, you can browse, create, and edit database objects. You can use the SQL Worksheet to run SQL statements and scripts. Using SQL Developer, you can edit and debug PL/SQL. SQL Developer enables you to create and save your own special set of reports for repeated use.

Index