Design Considerations for PL/SQL Code

Objectives

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

- Create standard constants and exceptions
- Write and call local subprograms
- Control the run-time privileges of a subprogram
- Perform autonomous transactions
- Pass parameters by reference using a `NOCOPY` hint
- Use the `PARALLEL ENABLE` hint for optimization
- Use the cross-session PL/SQL function result cache
- Use the `DETERMINISTIC` clause with functions
- Use bulk binding and the `RETURNING` clause with DML

Lesson Agenda

- Standardizing constants and exceptions, using local subprograms, controlling the run-time privileges of a subprogram, and performing autonomous transactions
- Using the `NOCOPY` and the `PARALLEL ENABLE` hints, the cross-session PL/SQL function result cache, and the `DETERMINISTIC` clause
- Using bulk binding and the `RETURNING` clause with DML

Standardizing Constants and Exceptions

Constants and exceptions are typically implemented using a bodiless package (that is, a package specification).

- Standardizing helps to:
  - Develop programs that are consistent
  - Promote a higher degree of code reuse
  - Ease code maintenance
  - Implement company standards across entire applications
- Start with standardization of:
  - Exception names
  - Constant definitions

Standardizing Exceptions

Create a standardized error-handling package that includes all named and programmer-defined exceptions to be used in the application.

```
CREATE OR REPLACE PACKAGE error_pkg IS
  e_fk_err         EXCEPTION;
  e_seq_nbr_err  EXCEPTION;
  PRAGMA EXCEPTION_INIT (e_fk_err, -2292);
  PRAGMA EXCEPTION_INIT (e_seq_nbr_err, -2277);
  ...
END error_pkg;
/
```

Standardizing Exception Handling

Consider writing a subprogram for common exception handling to:

- Display errors based on `SQLCODE` and `SQLERRM` values for exceptions
- Track run-time errors easily by using parameters in your code to identify:
  - The procedure in which the error occurred
  - The location (line number) of the error
  - `RAISE_APPLICATION_ERROR` using stack trace capabilities, with the third argument set to `TRUE`

Standardizing Constants

For programs that use local variables whose values should not change:

- Convert the variables to constants to reduce maintenance and debugging
- Create one central package specification and place all constants in it

```
CREATE OR REPLACE PACKAGE constant_pkg IS
  c_order_received CONSTANT VARCHAR(2) := 'OR';
  c_order_shipped  CONSTANT VARCHAR(2) := 'OS';
  c_min_sal        CONSTANT NUMBER(3)  := 900;
END constant_pkg;
```

Local Subprograms

A local subprogram is a PROCEDURE or FUNCTION defined at the end of the declarative section.

```
CREATE PROCEDURE employee_sal(p_id NUMBER) IS
   v_emp employees%ROWTYPE;
   FUNCTION tax(p_salary VARCHAR2) RETURN NUMBER IS
   BEGIN
     RETURN p_salary * 0.825;
   END tax;
BEGIN
   SELECT * INTO v_emp
   FROM EMPLOYEES WHERE employee_id = p_id;
   DBMS_OUTPUT.PUT_LINE('Tax: '|| tax(v_emp.salary));
END;
/
EXECUTE employee_sal(100)
```

```
PROCEDURE employee_sal(p_id Compiled.
anonymous block completed
Tax: 19800
```

Definer's Rights Versus Invoker's Rights

Definer's rights:

- Used prior to Oracle8*i*
- Programs execute with the privileges of the creating user.
- User does not require privileges on underlying objects that the procedure accesses. User requires privilege only to execute a procedure.

Invoker's rights:

- Introduced in Oracle8*i*
- Programs execute with the privileges of the calling user.
- User requires privileges on the underlying objects that the procedure accesses.

Specifying Invoker's Rights: Setting AUTHID to CURRENT_USER

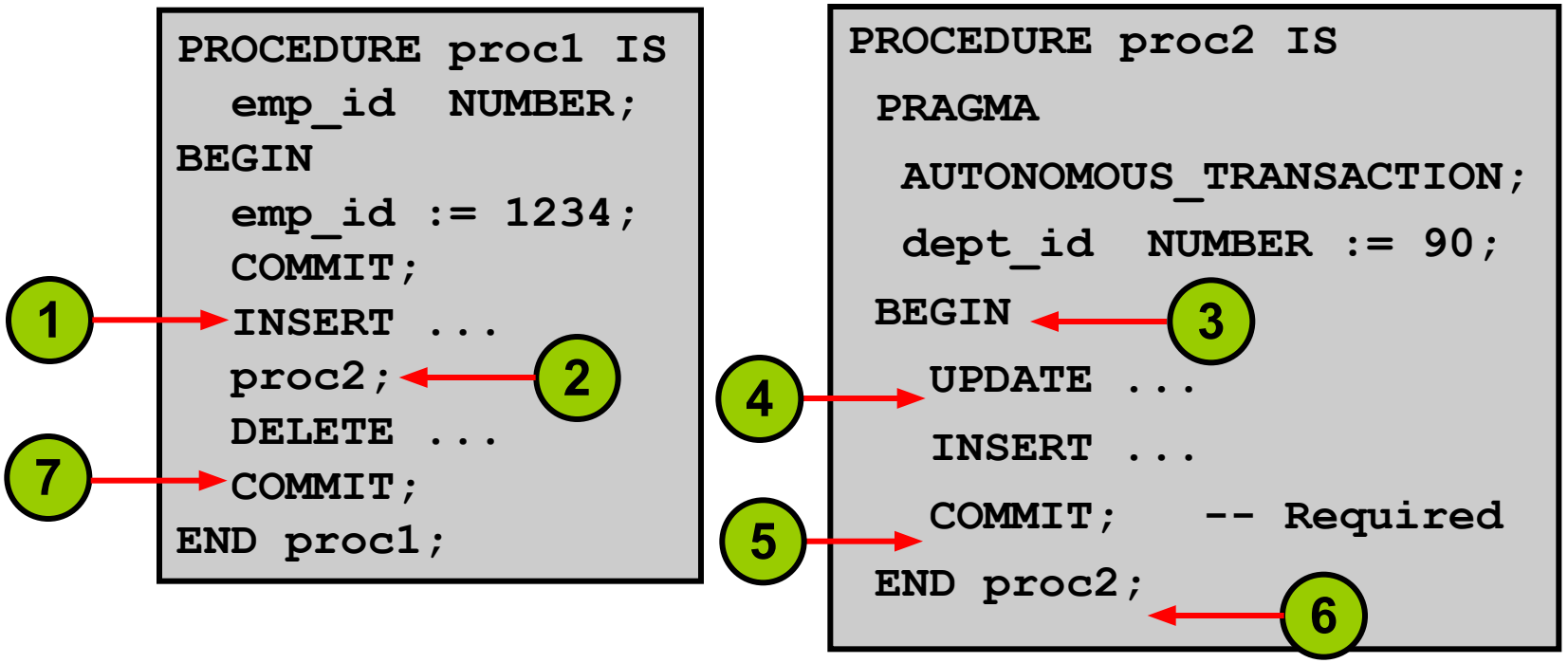
```
CREATE OR REPLACE PROCEDURE add_dept(
  p_id NUMBER, p_name VARCHAR2) AUTHID CURRENT_USER IS
BEGIN
  INSERT INTO departments
  VALUES (p_id, p_name, NULL, NULL);
END;
```

When used with stand-alone functions, procedures, or packages:

- Names used in queries, DML, Native Dynamic SQL, and DBMS_SQL package are resolved in the invoker's schema
- Calls to other packages, functions, and procedures are resolved in the definer's schema

Autonomous Transactions

- Are independent transactions started by another main transaction
- Are specified with `PRAGMA AUTONOMOUS_TRANSACTION`



Features of Autonomous Transactions

- Are independent of the main transaction
- Suspend the calling transaction until the autonomous transactions are completed
- Are not nested transactions
- Do not roll back if the main transaction rolls back
- Enable the changes to become visible to other transactions upon a commit
- Are started and ended by individual subprograms and not by nested or anonymous PL/SQL blocks

Using Autonomous Transactions: Example

```
PROCEDURE bank_trans(p_cardnbr NUMBER, p_loc NUMBER) IS
BEGIN
  log_usage(p_cardnbr, p_loc);
  INSERT INTO txn VALUES (9001, 1000,...);
END bank_trans;
```

```
PROCEDURE log_usage (p_card_id NUMBER, p_loc NUMBER)
IS
  PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
  INSERT INTO usage        -- usage is an existing table
  VALUES (p_card_id, p_loc);
  COMMIT;
END log_usage;
```

Lesson Agenda

- Standardizing constants and exceptions, using local subprograms, controlling the run-time privileges of a subprogram, and performing autonomous transactions
- Using the `NOCOPY` and the `PARALLEL ENABLE` hints, the cross-session PL/SQL function result cache, and the `DETERMINISTIC` clause
- Using bulk binding and the `RETURNING` clause with DML

Using the NOCOPY Hint

- Allows the PL/SQL compiler to pass OUT and IN OUT parameters by reference rather than by value
- Enhances performance by reducing overhead when passing parameters

```
DECLARE
  TYPE          rec_emp_type IS TABLE OF
   employees%ROWTYPE;
  rec_emp  rec_emp_type;
  PROCEDURE populate(p_tab IN OUT NOCOPY emptabtype)IS
    BEGIN
    . . .
    END;
BEGIN
  populate(rec_emp);
END;
/
```

Effects of the NOCOPY Hint

- If the subprogram exits with an exception that is not handled:
  - You cannot rely on the values of the actual parameters passed to a `NOCOPY` parameter
  - Any incomplete modifications are not “rolled back”
- The remote procedure call (RPC) protocol enables you to pass parameters only by value.

When Does the PL/SQL Compiler Ignore the NOCOPY Hint?

The NOCOPY hint has no effect if:

- The actual parameter:
  - Is an element of an index-by table
  - Is constrained (for example, by scale or NOT NULL)
  - And formal parameter are records, where one or both records were declared by using %ROWTYPE or %TYPE, and constraints on corresponding fields in the records differ
  - Requires an implicit data type conversion
- The subprogram is involved in an external or remote procedure call

Using the PARALLEL_ENABLE Hint

- Can be used in functions as an optimization hint
- Indicates that a function can be used in a parallelized query or parallelized DML statement

```
CREATE OR REPLACE FUNCTION f2 (p_p1 NUMBER)
  RETURN NUMBER PARALLEL_ENABLE IS
BEGIN
  RETURN p_p1 * 2;
END f2;
```

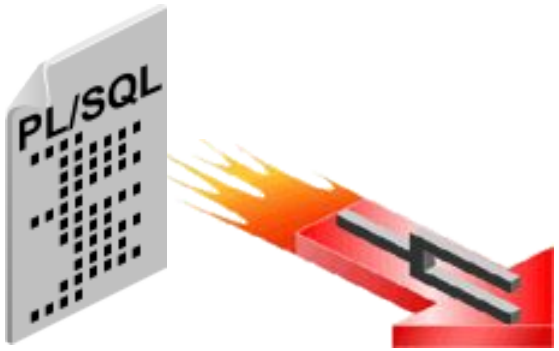
Using the Cross-Session PL/SQL Function Result Cache

- Each time a result-cached PL/SQL function is called with different parameter values, those parameters and their results are stored in cache.
- The function result cache is stored in a shared global area (SGA), making it available to any session that runs your application.
- Subsequent calls to the same function with the same parameters uses the result from cache.
- Performance and scalability are improved.
- This feature is used with functions that are called frequently and dependent on information that changes infrequently.

Enabling Result-Caching for a Function

You can make a function result-cached as follows:

- Include the `RESULT_CACHE` clause in the following:
  - The function declaration
  - The function definition
- Include an optional `RELIES_ON` clause to specify any tables or views on which the function results depend.



Declaring and Defining a Result-Cached Function: Example

```
CREATE OR REPLACE FUNCTION emp_hire_date (p_emp_id
    NUMBER) RETURN VARCHAR
 RESULT_CACHE RELIES_ON (employees) IS
  v_date_hired DATE;
BEGIN
  SELECT hire_date INTO v_date_hired
  FROM HR.Employees
  WHERE Employee_ID = p_emp_ID;
  RETURN to_char(v_date_hired);
END;
```

Using the `DETERMINISTIC` Clause with Functions

- Specify `DETERMINISTIC` to indicate that the function returns the same result value whenever it is called with the same values for its arguments.
- This helps the optimizer avoid redundant function calls.
- If a function was called previously with the same arguments, the optimizer can elect to use the previous result.
- Do not specify `DETERMINISTIC` for a function whose result depends on the state of session variables or schema objects.

Lesson Agenda

- Standardizing constants and exceptions, using local subprograms, controlling the run-time privileges of a subprogram, and performing autonomous transactions
- Using the `NOCOPY` and the `PARALLEL ENABLE` hints, the cross-session PL/SQL function result cache, and the `DETERMINISTIC` clause
- Using bulk binding and the `RETURNING` clause with DML

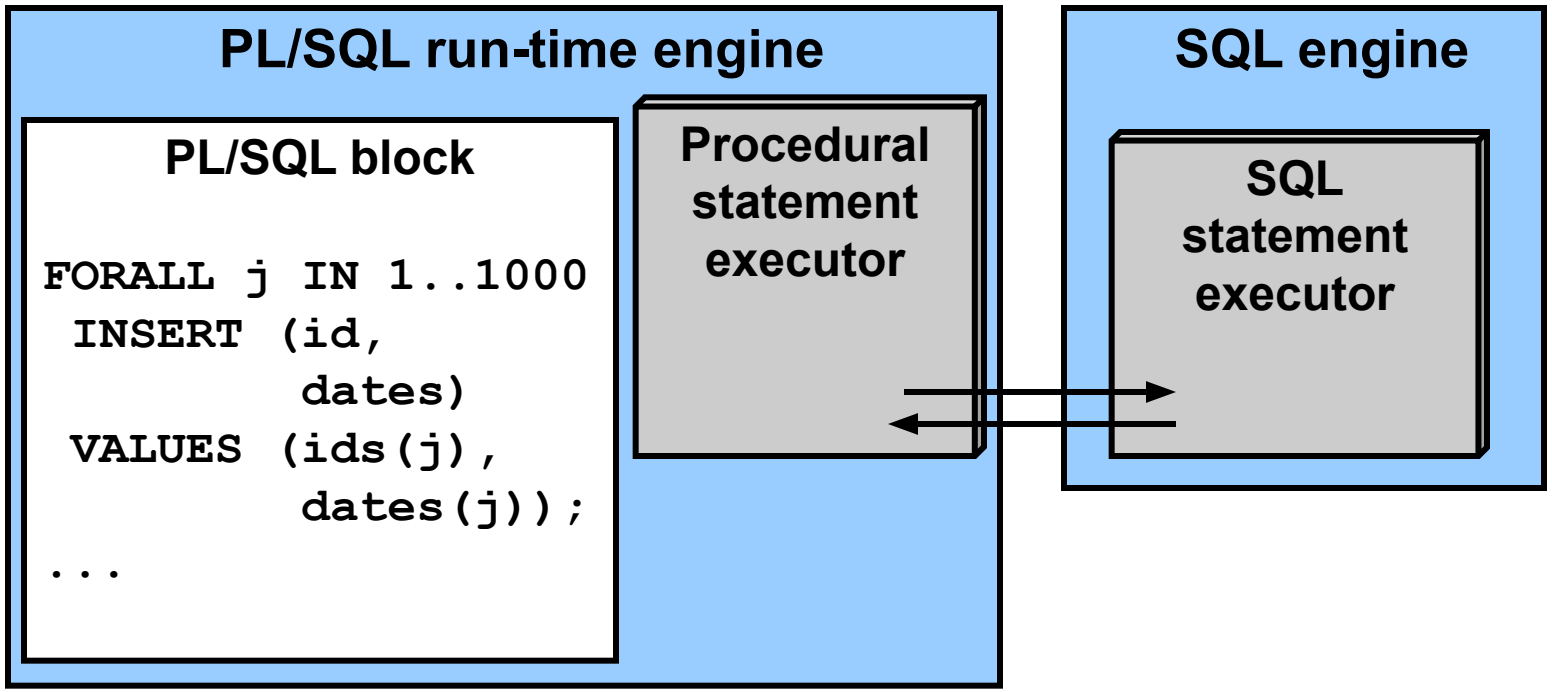
Using the RETURNING Clause

- Improves performance by returning column values with INSERT, UPDATE, and DELETE statements
- Eliminates the need for a SELECT statement

```
CREATE PROCEDURE update_salary(p_emp_id NUMBER) IS
  v_name    employees.last_name%TYPE;
  v_new_sal employees.salary%TYPE;
BEGIN
  UPDATE employees
    SET salary = salary * 1.1
  WHERE employee_id = p_emp_id
  RETURNING last_name, salary INTO name, v_new_sal;
END update_salary;
/
```

Bulk Binding

Binds whole arrays of values in a single operation, rather than using a loop to perform a `FETCH`, `INSERT`, `UPDATE`, and `DELETE` operation multiple times



Using Bulk Binding: Syntax and Keywords

- The `FORALL` keyword instructs the *PL/SQL engine* to bulk bind input collections before sending them to the SQL engine.

```
FORALL index IN lower_bound .. upper_bound
  [SAVE EXCEPTIONS]
  sql_statement;
```

- The `BULK COLLECT` keyword instructs the *SQL engine* to bulk bind output collections before returning them to the PL/SQL engine.

```
... BULK COLLECT INTO
      collection_name[,collection_name] ...
```

Bulk Binding FORALL: Example

```
CREATE PROCEDURE raise_salary(p_percent NUMBER) IS
  TYPE numlist_type IS TABLE OF NUMBER
     INDEX BY BINARY_INTEGER;
  v_id  numlist_type; -- collection
BEGIN
 v_id(1):= 100; v_id(2):= 102; v_id(3):= 104; v_id(4) := 110;
 -- bulk-bind the PL/SQL table
  FORALL i IN v_id.FIRST .. v_id.LAST
    UPDATE employees
      SET salary = (1 + p_percent/100) * salary
      WHERE employee_id = v_id(i);
END;
/
```

```
EXECUTE raise_salary(10)

PL/SQL procedure successfully completed.
```

Using BULK COLLECT INTO with Queries

The SELECT statement has been enhanced to support the BULK COLLECT INTO syntax.

```
CREATE PROCEDURE get_departments(p_loc NUMBER) IS
  TYPE dept_tab_type IS
    TABLE OF departments%ROWTYPE;
  v_depts dept_tab_type;
BEGIN
  SELECT * BULK COLLECT INTO v_depts
  FROM departments
  WHERE location_id = p_loc;
  FOR i IN 1 .. v_depts.COUNT LOOP
    DBMS_OUTPUT.PUT_LINE(v_depts(i).department_id
     ||' '|| v_depts(i).department_name);
  END LOOP;
END;
```

Using BULK COLLECT INTO with Cursors

The FETCH statement has been enhanced to support the BULK COLLECT INTO syntax.

```
CREATE PROCEDURE get_departments(p_loc NUMBER) IS
  CURSOR cur_dept IS
    SELECT * FROM departments
    WHERE location_id = p_loc;
  TYPE dept_tab_type IS TABLE OF cur_dept%ROWTYPE;
  v_depts dept_tab_type;
BEGIN
  OPEN cur_dept;
  FETCH cur_dept BULK COLLECT INTO v_depts;
  CLOSE cur_dept;
 FOR i IN 1 .. v_depts.COUNT LOOP
    DBMS_OUTPUT.PUT_LINE(v_depts(i).department_id
     ||' '|| v_depts(i).department_name);
  END LOOP;
END;
```

Using BULK COLLECT INTO with a RETURNING Clause

```
CREATE PROCEDURE raise_salary(p_rate NUMBER) IS
   TYPE emplist_type IS TABLE OF NUMBER;
   TYPE numlist_type IS TABLE OF employees.salary%TYPE
     INDEX BY BINARY_INTEGER;
   v_emp_ids  emplist_type :=
   emplist_type(100,101,102,104);
   v_new_sals numlist_type;
BEGIN
  FORALL i IN v_emp_ids.FIRST .. v_emp_ids.LAST
    UPDATE employees
      SET commission_pct = p_rate * salary
    WHERE employee_id = v_emp_ids(i)
    RETURNING salary BULK COLLECT INTO v_new_sals;
  FOR i IN 1 ..  v_new_sals.COUNT LOOP ...
END;
```

FORALL Support for Sparse Collections

```
-- The new INDICES OF syntax allows the bound arrays
-- themselves to be sparse.

FORALL index_name IN INDICES OF sparse_array_name
      BETWEEN LOWER_BOUND AND UPPER_BOUND -- optional
      SAVE EXCEPTIONS  -- optional, but recommended
      INSERT INTO table_name VALUES
    sparse_array(index_name);
```

```
-- The new VALUES OF syntax lets you indicate a subset
-- of the binding arrays.

FORALL index_name IN VALUES OF index_array_name
      SAVE EXCEPTIONS -- optional,but recommended
      INSERT INTO table_name VALUES
    binding_array_name(index_name);
```

Using Bulk Binds in Sparse Collections

The typical application for this feature is an order entry and order processing system where:

- Users enter orders through the Web
- Orders are placed in a staging table before validation
- Data is later validated based on complex business rules (usually implemented programmatically using PL/SQL)
- Invalid orders are separated from valid ones
- Valid orders are inserted into a permanent table for processing

Using Bulk Bind with Index Array

```
CREATE OR REPLACE PROCEDURE ins_emp2 AS
  TYPE emptab_type IS TABLE OF employees%ROWTYPE;
  v_emp emptab_type;
  TYPE values_of_tab_type IS TABLE OF PLS_INTEGER
      INDEX BY PLS_INTEGER;
  v_num   values_of_tab_type;
    . . .
BEGIN
    . . .
    FORALL k IN VALUES OF v_num
    INSERT INTO new_employees VALUES v_emp(k);
END;
```

Quiz

The `NOCOPY` hint allows the PL/SQL compiler to pass `OUT` and `IN OUT` parameters by reference rather than by value. This enhances performance by reducing overhead when passing parameters

1. True
2. False

Summary

In this lesson, you should have learned how to:

- Create standard constants and exceptions
- Write and call local subprograms
- Control the run-time privileges of a subprogram
- Perform autonomous transactions
- Pass parameters by reference using a `NOCOPY` hint
- Use the `PARALLEL ENABLE` hint for optimization
- Use the cross-session PL/SQL function result cache
- Use the `DETERMINISTIC` clause with functions
- Use bulk binding and the `RETURNING` clause with DML

Practice: Overview

This practice covers the following topics:

- Creating a package that uses bulk fetch operations
- Creating a local subprogram to perform an autonomous transaction to audit a business operation