7

Design Considerations for PL/SQL Code

Objectives

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

- Use package specifications to create standard constants and exceptions
- Write and call local subprograms
- Set the `AUTHID` directive to control the run-time privileges of a subprogram
- Execute subprograms to perform autonomous transactions
- Use bulk binding and the `RETURNING` clause with DML
- Pass parameters by reference using a `NOCOPY` hint
- Use the `PARALLEL ENABLE` hint for optimization

Standardizing Constants and Exceptions

Constants and exceptions are typically implemented using a bodiless package (that is, in 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
  fk_err        EXCEPTION;
  seq_nbr_err   EXCEPTION;
  PRAGMA EXCEPTION_INIT (fk_err, -2292);
  PRAGMA EXCEPTION_INIT (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 in the declarative section.

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

- The local subprogram must be defined at the end of the declarative section.

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

Set AUTHID to CURRENT_USER:

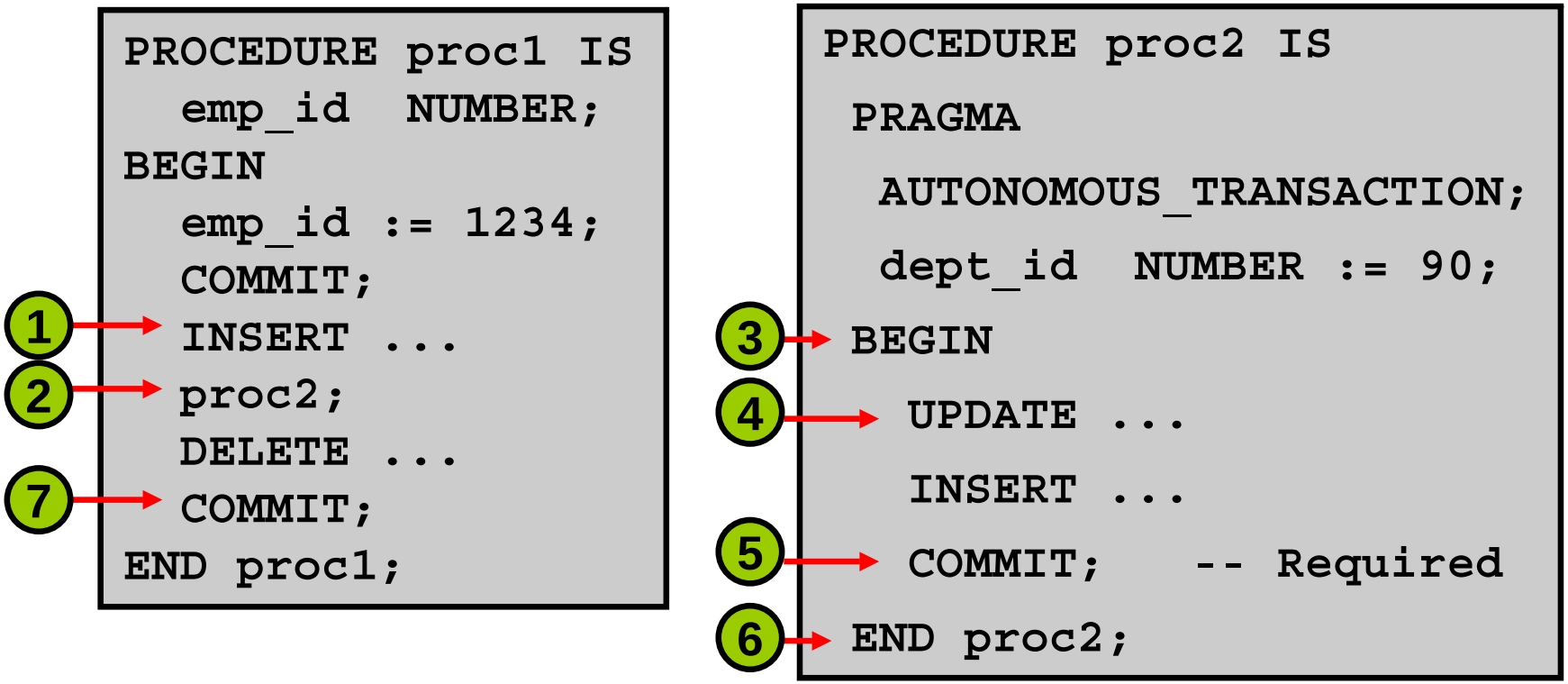
```
CREATE OR REPLACE PROCEDURE add_dept(
  id NUMBER, name VARCHAR2) AUTHID CURRENT_USER IS
BEGIN
  INSERT INTO departments
  VALUES (id,name,NULL,NULL);
END;
```

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

Autonomous transactions:

- Are independent of the main transaction
- Suspend the calling transaction until it is 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 demarcated (started and ended) by individual subprograms and not by nested or anonymous PL/SQL blocks

Using Autonomous Transactions

Example:

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

```
PROCEDURE log_usage (card_id NUMBER, loc NUMBER)
IS
  PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
  INSERT INTO usage
  VALUES (card_id, loc);
  COMMIT;
END log_usage;
```

RETURNING Clause

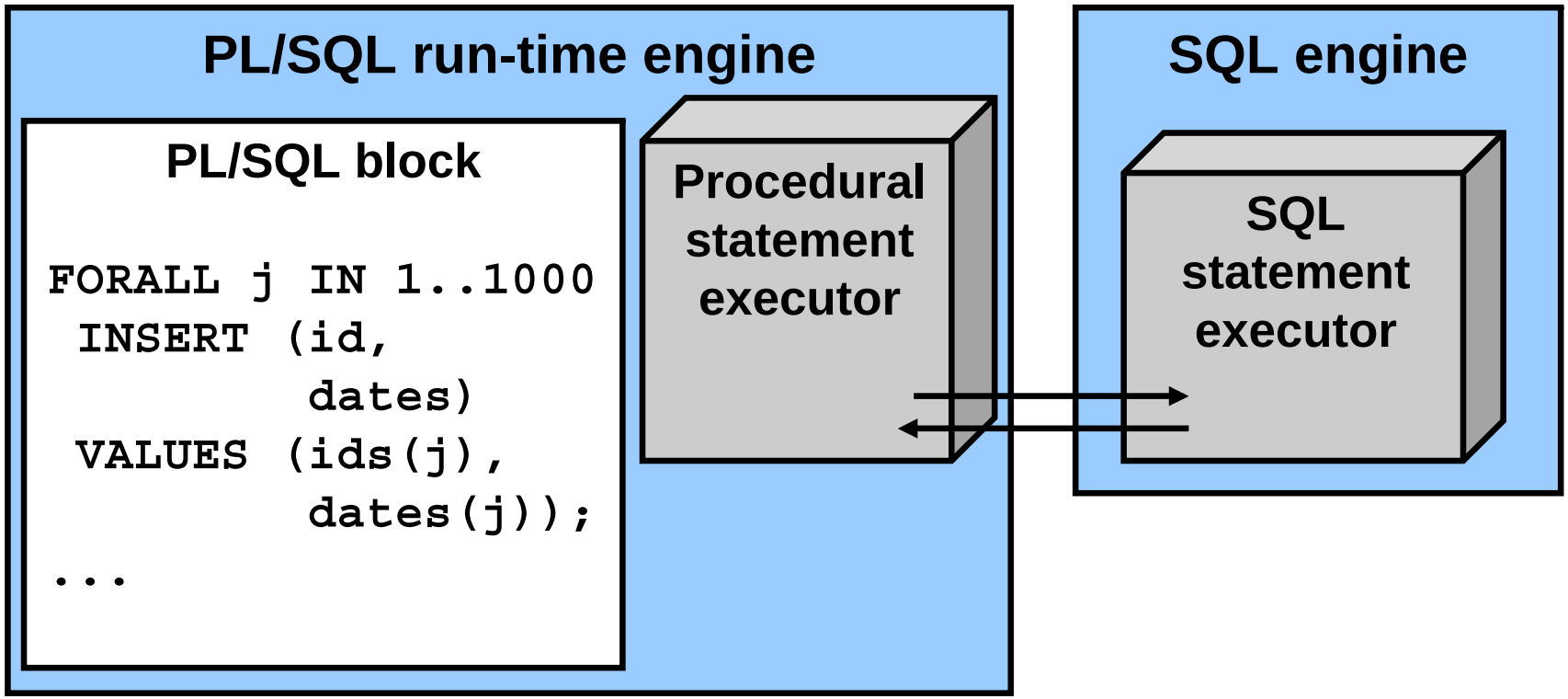
The RETURNING clause:

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

```
CREATE PROCEDURE update_salary(emp_id NUMBER) IS
  name    employees.last_name%TYPE;
  new_sal employees.salary%TYPE;
BEGIN
  UPDATE employees
    SET salary = salary * 1.1
  WHERE employee_id = emp_id
  RETURNING last_name, salary INTO name, 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

Keywords to support bulk binding:

- 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(percent NUMBER) IS
  TYPE numlist IS TABLE OF NUMBER
    INDEX BY BINARY_INTEGER;
  id  numlist;
BEGIN
 id(1) := 100; id(2) := 102;
 id(3) := 104; id(4) := 110;
 -- bulk-bind the PL/SQL table
  FORALL i IN id.FIRST .. id.LAST
    UPDATE employees
      SET salary = (1 + percent/100) * salary
      WHERE manager_id = id(i);
END;
/
```

```
EXECUTE raise_salary(10)
```

Using BULK COLLECT INTO with Queries

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

Example:

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

Example:

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

Using BULK COLLECT INTO with a RETURNING Clause

Example:

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

Using the NOCOPY Hint

The NOCOPY hint:

- Is a request to 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 emptabtype IS TABLE OF employees%ROWTYPE;
  emp_tab emptabtype;
  PROCEDURE populate(tab IN OUT NOCOPY emptabtype)
  IS BEGIN ... END;
BEGIN
  populate(emp_tab);
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.

NOCOPY Hint Can Be Ignored

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

PARALLEL_ENABLE Hint

The PARALLEL_ENABLE hint:

- Can be used in functions as an optimization hint

```
CREATE OR REPLACE FUNCTION f2 (p1 NUMBER)
  RETURN NUMBER PARALLEL_ENABLE IS
BEGIN
  RETURN p1 * 2;
END f2;
```

- Indicates that a function can be used in a parallelized query or parallelized DML statement

Summary

In this lesson, you should have learned how to:

- Create standardized constants and exceptions using packages
- Develop and invoke local subprograms
- Control the run-time privileges of a subprogram by setting the `AUTHID` directive
- Execute autonomous transactions
- Use the `RETURNING` clause with DML statements, and bulk binding collections with the `FORALL` and `BULK COLLECT INTO` clauses
- Pass parameters by reference using a `NOCOPY` hint
- Enable optimization with `PARALLEL ENABLE` hints

Practice 7: 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
- Testing `AUTHID` functionality