11 PL/SQL Packages

This chapter explains how to bundle related PL/SQL code and data into a package, whose contents are available to many applications.

Topics

What is a Package?

A **package** is a schema object that groups logically related PL/SQL types, variables, constants, subprograms, cursors, and exceptions. A package is compiled and stored in the database, where many applications can share its contents.

A package always has a **specification**, which declares the **public items** that can be referenced from outside the package.

If the public items include cursors or subprograms, then the package must also have a **body**. The body must define queries for public cursors and code for public subprograms. The body can also declare and define **private items** that cannot be referenced from outside the package, but are necessary for the internal workings of the package. Finally, the body can have an **initialization part**, whose statements initialize variables and do other one-time setup steps, and an exception-handling part. You can change the body without changing the specification or the references to the public items; therefore, you can think of the package body as a black box.

In either the package specification or package body, you can map a package subprogram to an external Java, JavaScript, or C subprogram by using a **call specification**, which maps the external subprogram name, parameter types, and return type to their SQL counterparts.

The `AUTHID` **clause** of the package specification determines whether the subprograms and cursors in the package run with the privileges of their definer (the default) or invoker, and whether their unqualified references to schema objects are resolved in the schema of the definer or invoker.

The `ACCESSIBLE BY` **clause** of the package specification lets you specify a white list of PL/SQL units that can access the package. You use this clause in situations like these:

- You implement a PL/SQL application as several packages—one package that provides the application programming interface (API) and helper packages to do the work. You want clients to have access to the API, but not to the helper packages. Therefore, you omit the `ACCESSIBLE BY` clause from the API package specification and include it in each helper package specification, where you specify that only the API package can access the helper package.
- You create a utility package to provide services to some, but not all, PL/SQL units in the same schema. To restrict use of the package to the intended units, you list them in the `ACCESSIBLE BY` clause in the package specification.

> **See Also:**
>
> - "Package Specification" for more information about the package specification
> - "Package Body" for more information about the package body
> - "Function Declaration and Definition"
> - "Procedure Declaration and Definition"
> - "Invoker's Rights and Definer's Rights (AUTHID Property)"

Reasons to Use Packages

Packages support the development and maintenance of reliable, reusable code with the following features:

- **Modularity**

  Packages let you encapsulate logically related types, variables, constants, subprograms, cursors, and exceptions in named PL/SQL modules. You can make each package easy to understand, and make the interfaces between packages simple, clear, and well defined. This practice aids application development.

- **Easier Application Design**

  When designing an application, all you need initially is the interface information in the package specifications. You can code and compile specifications without their bodies. Next, you can compile standalone subprograms that reference the packages. You need not fully define the package bodies until you are ready to complete the application.

- **Hidden Implementation Details**

  Packages let you share your interface information in the package specification, and hide the implementation details in the package body. Hiding the implementation details in the body has these advantages:

  - You can change the implementation details without affecting the application interface.
  - Application users cannot develop code that depends on implementation details that you might want to change.

- **Added Functionality**

  Package public variables and cursors can persist for the life of a session. They can be shared by all subprograms that run in the environment. They let you maintain data across transactions without storing it in the database. (For the situations in which package public variables and cursors do not persist for the life of a session, see "Package State".)

- **Better Performance**

  The first time you invoke a package subprogram, Oracle Database loads the whole package into memory. Subsequent invocations of other subprograms in same the package require no disk I/O.

  Packages prevent cascading dependencies and unnecessary recompiling. For example, if you change the body of a package function, Oracle Database does not recompile other subprograms that invoke the function, because these subprograms depend only on the parameters and return value that are declared in the specification.

- **Easier to Grant Roles**

  You can grant roles on the package, instead of granting roles on each object in the package.

> **Note:**
>
> You cannot reference host variables from inside a package.

Package Specification

A **package specification** declares **public items**. The scope of a public item is the schema of the package. A public item is visible everywhere in the schema. To reference a public item that is in scope but not visible, qualify it with the package name. (For information about scope, visibility, and qualification, see "Scope and Visibility of Identifiers".)

Each public item declaration has all information needed to use the item. For example, suppose that a package specification declares the function `factorial` this way:

```
FUNCTION factorial (n INTEGER) RETURN INTEGER; -- returns n!
```

The declaration shows that `factorial` needs one argument of type `INTEGER` and returns a value of type `INTEGER`, which is invokers must know to invoke `factorial`. Invokers need not know how `factorial` is implemented (for example, whether it is iterative or recursive).

> **Note:**
>
> To restrict the use of your package to specified PL/SQL units, include the `ACCESSIBLE BY` clause in the package specification.

Topics

- Appropriate Public Items
- Creating Package Specifications

Appropriate Public Items

Appropriate public items are:

- Types, variables, constants, subprograms, cursors, and exceptions used by multiple subprograms

A type defined in a package specification is either a PL/SQL user-defined subtype (described in "User-Defined PL/SQL Subtypes") or a PL/SQL composite type (described in PL/SQL Collections and Records).

> **Note:**
>
> A PL/SQL composite type defined in a package specification is incompatible with an identically defined local or standalone type (see Example 6-37, Example 6-38, and Example 6-44).

- Associative array types of standalone subprogram parameters

  You cannot declare an associative array type at schema level. Therefore, to pass an associative array variable as a parameter to a standalone subprogram, you must declare the type of that variable in a package specification. Doing so makes the type available to both the invoked subprogram (which declares a formal parameter of that type) and to the invoking subprogram or anonymous block (which declares a variable of that type). See Example 11-2.
- Variables that must remain available between subprogram invocations in the same session
- Subprograms that read and write public variables ("get" and "set" subprograms)

  Provide these subprograms to discourage package users from reading and writing public variables directly.
- Subprograms that invoke each other

  You need not worry about compilation order for package subprograms, as you must for standalone subprograms that invoke each other.
- Overloaded subprograms

  Overloaded subprograms are variations of the same subprogram. That is, they have the same name but different formal parameters. For more information about them, see "Overloaded Subprograms".

> **Note:**
>
> You cannot reference remote package public variables, even indirectly. For example, if a subprogram refers to a package public variable, you cannot invoke the subprogram through a database link.

Creating Package Specifications

To create a package specification, use the "CREATE PACKAGE Statement".

Because the package specifications in Example 11-1 and Example 11-2 do not declare cursors or subprograms, the packages `trans_data` and `aa_pkg` do not need bodies.

Example 11-1 Simple Package Specification

In this example, the specification for the package `trans_data` declares two public types and three public variables.

```
CREATE OR REPLACE PACKAGE trans_data AUTHID DEFINER AS
  TYPE TimeRec IS RECORD (
    minutes SMALLINT,
    hours   SMALLINT);
  TYPE TransRec IS RECORD (
    category VARCHAR2(10),
    account  INT,
    amount   REAL,
    time_of  TimeRec);
  minimum_balance     CONSTANT REAL := 10.00;
  number_processed    INT;
  insufficient_funds  EXCEPTION;
  PRAGMA EXCEPTION_INIT(insufficient_funds, -4097);
END trans_data;
/
```

Example 11-2 Passing Associative Array to Standalone Subprogram

In this example, the specification for the package `aa_pkg` declares an associative array type, `aa_type`. Then, the standalone procedure `print_aa` declares a formal parameter of type `aa_type`. Next, the anonymous block declares a variable of type `aa_type`, populates it, and passes it to the procedure `print_aa`, which prints it.

```
CREATE OR REPLACE PACKAGE aa_pkg AUTHID DEFINER IS
  TYPE aa_type IS TABLE OF INTEGER INDEX BY VARCHAR2(15);
END;
/
CREATE OR REPLACE PROCEDURE print_aa (
  aa aa_pkg.aa_type
) AUTHID DEFINER IS
  i  VARCHAR2(15);
BEGIN
  i := aa.FIRST;

  WHILE i IS NOT NULL LOOP
    DBMS_OUTPUT.PUT_LINE (aa(i) || '  ' || i);
    i := aa.NEXT(i);
  END LOOP;
END;
/
DECLARE
  aa_var  aa_pkg.aa_type;
BEGIN
  aa_var('zero') := 0;
  aa_var('one') := 1;
  aa_var('two') := 2;
  print_aa(aa_var);
END;
/
```

Result:

```
1  one
2  two
0  zero
```

Package Body

If a package specification declares cursors or subprograms, then a package body is required; otherwise, it is optional. The package body and package specification must be in the same schema.

Every cursor or subprogram declaration in the package specification must have a corresponding definition in the package body. The headings of corresponding subprogram declarations and definitions must match word for word, except for white space.

To create a package body, use the "CREATE PACKAGE BODY Statement".

The cursors and subprograms declared in the package specification and defined in the package body are public items that can be referenced from outside the package. The package body can also declare and define **private items** that cannot be referenced from outside the package, but are necessary for the internal workings of the package.

Finally, the body can have an **initialization part**, whose statements initialize public variables and do other one-time setup steps. The initialization part runs only the first time the package is referenced. The initialization part can include an exception handler.

You can change the package body without changing the specification or the references to the public items.

Example 11-3 Matching Package Specification and Body

In this example, the headings of the corresponding subprogram declaration and definition do not match word for word; therefore, PL/SQL raises an exception, even though `employees.hire_date%TYPE` is `DATE`.

```
CREATE PACKAGE emp_bonus AS
  PROCEDURE calc_bonus (date_hired employees.hire_date%TYPE);
END emp_bonus;
/
CREATE PACKAGE BODY emp_bonus AS
  -- DATE does not match employees.hire_date%TYPE
  PROCEDURE calc_bonus (date_hired DATE) IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE
      ('Employees hired on ' || date_hired || ' get bonus.');
  END;
END emp_bonus;
/
```

Result:

```
Warning: Package Body created with compilation errors.
```

Show errors (in SQL*Plus):

```
SHOW ERRORS
```

Result:

```
Errors for PACKAGE BODY EMP_BONUS:

LINE/COL ERROR
-------- -----------------------------------------------------------------
2/13     PLS-00323: subprogram or cursor 'CALC_BONUS' is declared in a
         package specification and must be defined in the package body
```

Correct problem:

```
CREATE OR REPLACE PACKAGE BODY emp_bonus AS
  PROCEDURE calc_bonus
    (date_hired employees.hire_date%TYPE) IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE
      ('Employees hired on ' || date_hired || ' get bonus.');
  END;
END emp_bonus;
/
```

Result:

```
Package body created.
```

Package Instantiation and Initialization

When a session references a package item, Oracle Database instantiates the package for that session. Every session that references a package has its own instantiation of that package.

When Oracle Database instantiates a package, it initializes it. Initialization includes whichever of the following are applicable:

- Assigning initial values to public constants
- Assigning initial values to public variables whose declarations specify them
- Executing the initialization part of the package body

Package State

The values of the variables, constants, and cursors that a package declares (in either its specification or body) comprise its **package state**.

If a PL/SQL package declares at least one variable, constant, or cursor, then the package is **stateful**; otherwise, it is **stateless**.

Each session that references a package item has its own instantiation of that package. If the package is stateful, the instantiation includes its state.

The package state persists for the life of a session, except in these situations:

- The package is `SERIALLY_REUSABLE`.
- The package body is recompiled.

  If the body of an instantiated, stateful package is recompiled (either explicitly, with the "ALTER PACKAGE Statement", or implicitly), the next invocation of a subprogram in the package causes Oracle Database to discard the existing package state and raise the exception ORA-04068.

  After PL/SQL raises the exception, a reference to the package causes Oracle Database to re-instantiate the package, which re-initializes it. Therefore, previous changes to the package state are lost.
- Any of the session's instantiated packages are invalidated and revalidated.

  All of a session's package instantiations (including package states) can be lost if any of the session's instantiated packages are invalidated and revalidated.

Oracle Database treats a package as stateless if its state is constant for the life of a session (or longer). This is the case for a package whose items are all compile-time constants.

A **compile-time constant** is a constant whose value the PL/SQL compiler can determine at compilation time. A constant whose initial value is a literal is always a compile-time constant. A constant whose initial value is not a literal, but which the optimizer reduces to a literal, is also a compile-time constant. Whether the PL/SQL optimizer can reduce a nonliteral expression to a literal depends on optimization level. Therefore, a package that is stateless when compiled at one optimization level might be stateful when compiled at a different optimization level.

Starting with Oracle Database 19c, Release Update 19.23, the initialization parameter SESSION_EXIT_ON_PACKAGE_STATE_ERROR allows you to specify behavior in the event package state is invalidated. When a stateful PL/SQL package undergoes modification, the sessions that have an active instantiation of the package receive the following error when they attempt to run it:

```
ORA-04068: existing state of package has been discarded
```

When SESSION_EXIT_ON_PACKAGE_STATE_ERROR is set to TRUE, the session immediately exits instead of just raising ORA-04068. This can be advantageous because many applications are better equipped to handle a session being discarded, simplifying recovery.

> **See Also:**
>
> - "SERIALLY_REUSABLE Packages"
> - "Package Instantiation and Initialization" for information about initialization
> - *Oracle Database Development Guide* for information about invalidation and revalidation of schema objects
> - "PL/SQL Optimizer" for information about the optimizer
> - *Oracle Database Reference* for more information about SESSION_EXIT_ON_PACKAGE_STATE_ERROR

SERIALLY_REUSABLE Packages

SERIALLY_REUSABLE packages let you design applications that manage memory better for scalability.

If a package is not SERIALLY_REUSABLE, its package state is stored in the user global area (UGA) for each user. Therefore, the amount of UGA memory needed increases linearly with the number of users, limiting scalability. The package state can persist for the life of a session, locking UGA memory until the session ends. In some applications, such as Oracle Office, a typical session lasts several days.

If a package is SERIALLY_REUSABLE, its package state is stored in a work area in a small pool in the system global area (SGA). The package state persists only for the life of a server call. After the server call, the work area returns to the pool. If a subsequent server call references the package, then Oracle Database reuses an instantiation from the pool. Reusing an instantiation re-initializes it; therefore, changes made to the package state in previous server calls are invisible. (For information about initialization, see "Package Instantiation and Initialization".)

> **Note:**
>
> Trying to access a SERIALLY_REUSABLE package from a database trigger, or from a PL/SQL subprogram invoked by a SQL statement, raises an error.

Topics

- Creating SERIALLY_REUSABLE Packages

- SERIALLY_REUSABLE Package Work Unit
- Explicit Cursors in SERIALLY_REUSABLE Packages

Creating SERIALLY_REUSABLE Packages

To create a `SERIALLY_REUSABLE` package, include the `SERIALLY_REUSABLE` pragma in the package specification and, if it exists, the package body.

Example 11-4 creates two very simple `SERIALLY_REUSABLE` packages, one with only a specification, and one with both a specification and a body.

> **See Also:**
>
> "SERIALLY_REUSABLE Pragma"

Example 11-4 Creating SERIALLY_REUSABLE Packages

```
-- Create bodiless SERIALLY_REUSABLE package:

CREATE OR REPLACE PACKAGE bodiless_pkg AUTHID DEFINER IS
  PRAGMA SERIALLY_REUSABLE;
  n NUMBER := 5;
END;
/

-- Create SERIALLY_REUSABLE package with specification and body:

CREATE OR REPLACE PACKAGE pkg AUTHID DEFINER IS
  PRAGMA SERIALLY_REUSABLE;
  n NUMBER := 5;
END;
/

CREATE OR REPLACE PACKAGE BODY pkg IS
  PRAGMA SERIALLY_REUSABLE;
BEGIN
  n := 5;
END;
/
```

SERIALLY_REUSABLE Package Work Unit

For a `SERIALLY_REUSABLE` package, the work unit is a server call.

You must use its public variables only within the work unit.

> **Note:**
>
> If you make a mistake and depend on the value of a public variable that was set in a previous work unit, then your program can fail. PL/SQL cannot check for such cases.

After the work unit (server call) of a SERIALLY_REUSABLE package completes, Oracle Database does the following:

- Closes any open cursors.
- Frees some nonreusable memory (for example, memory for collection and long VARCHAR2 variables)
- Returns the package instantiation to the pool of reusable instantiations kept for this package.

Example 11-5 Effect of SERIALLY_REUSABLE Pragma

In this example, the bodiless packages pkg and sr_pkg are the same, except that sr_pkg is SERIALLY_REUSABLE and pkg is not. Each package declares public variable n with initial value 5. Then, an anonymous block changes the value of each variable to 10. Next, another anonymous block prints the value of each variable. The value of pkg.n is still 10, because the state of pkg persists for the life of the session. The value of sr_pkg.n is 5, because the state of sr_pkg persists only for the life of the server call.

```
CREATE OR REPLACE PACKAGE pkg IS
  n NUMBER := 5;
END pkg;
/

CREATE OR REPLACE PACKAGE sr_pkg IS
  PRAGMA SERIALLY_REUSABLE;
  n NUMBER := 5;
END sr_pkg;
/

BEGIN
  pkg.n := 10;
  sr_pkg.n := 10;
END;
/

BEGIN
  DBMS_OUTPUT.PUT_LINE('pkg.n: ' || pkg.n);
  DBMS_OUTPUT.PUT_LINE('sr_pkg.n: ' || sr_pkg.n);
END;
/
```

Result:

```
pkg.n: 10
sr_pkg.n: 5
```

Explicit Cursors in SERIALLY_REUSABLE Packages

An explicit cursor in a SERIALLY_REUSABLE package remains open until either you close it or its work unit (server call) ends. To re-open the cursor, you must make a new server call. A server call can be different from a subprogram invocation, as Example 11-6 shows.

In contrast, an explicit cursor in a package that is not SERIALLY_REUSABLE remains open until you either close it or disconnect from the session.

Example 11-6 Cursor in SERIALLY_REUSABLE Package Open at Call Boundary

```
DROP TABLE people;
CREATE TABLE people (name VARCHAR2(20));
```

```
INSERT INTO people (name) VALUES ('John Smith');
INSERT INTO people (name) VALUES ('Mary Jones');
INSERT INTO people (name) VALUES ('Joe Brown');
INSERT INTO people (name) VALUES ('Jane White');

CREATE OR REPLACE PACKAGE sr_pkg IS
  PRAGMA SERIALLY_REUSABLE;
  CURSOR c IS SELECT name FROM people;
END sr_pkg;
/

CREATE OR REPLACE PROCEDURE fetch_from_cursor IS
  v_name  people.name%TYPE;
BEGIN
  IF sr_pkg.c%ISOPEN THEN
    DBMS_OUTPUT.PUT_LINE('Cursor is open.');
  ELSE
    DBMS_OUTPUT.PUT_LINE('Cursor is closed; opening now.');
    OPEN sr_pkg.c;
  END IF;

  FETCH sr_pkg.c INTO v_name;
  DBMS_OUTPUT.PUT_LINE('Fetched: ' || v_name);

  FETCH sr_pkg.c INTO v_name;
    DBMS_OUTPUT.PUT_LINE('Fetched: ' || v_name);
  END fetch_from_cursor;
/
```

First call to server:

```
BEGIN
  fetch_from_cursor;
  fetch_from_cursor;
END;
/
```

Result:

```
Cursor is closed; opening now.
Fetched: John Smith
Fetched: Mary Jones
Cursor is open.
Fetched: Joe Brown
Fetched: Jane White
```

New call to server:

```
BEGIN
  fetch_from_cursor;
  fetch_from_cursor;
END;
/
```

Result:

```
Cursor is closed; opening now.
Fetched: John Smith
Fetched: Mary Jones
Cursor is open.
```

```
Fetched: Joe Brown
Fetched: Jane White
```

Package Writing Guidelines

- Become familiar with the packages that Oracle Database supplies, and avoid writing packages that duplicate their features.

  For more information about the packages that Oracle Database supplies, see *Oracle Database PL/SQL Packages and Types Reference*.
- Keep your packages general so that future applications can reuse them.
- Design and define the package specifications before the package bodies.
- In package specifications, declare only items that must be visible to invoking programs.

  This practice prevents other developers from building unsafe dependencies on your implementation details and reduces the need for recompilation.

  If you change the package specification, you must recompile any subprograms that invoke the public subprograms of the package. If you change only the package body, you need not recompile those subprograms.
- Declare public cursors in package specifications and define them in package bodies, as in Example 11-7.

  This practice lets you hide cursors' queries from package users and change them without changing cursor declarations.
- Assign initial values in the initialization part of the package body instead of in declarations.

  This practice has these advantages:

  - The code for computing the initial values can be more complex and better documented.
  - If computing an initial value raises an exception, the initialization part can handle it with its own exception handler.
- If you implement a database application as several PL/SQL packages—one package that provides the API and helper packages to do the work, then make the helper packages available only to the API package, as in Example 11-8.

In Example 11-7, the declaration and definition of the cursor `c1` are in the specification and body, respectively, of the package `emp_stuff`. The cursor declaration specifies only the data type of the return value, not the query, which appears in the cursor definition (for complete syntax and semantics, see "Explicit Cursor Declaration and Definition").

Example 11-8 creates an API package and a helper package. Because of the `ACCESSIBLE BY` clause in the helper package specification, only the API package can access the helper package.

Example 11-7 Separating Cursor Declaration and Definition in Package

```
CREATE PACKAGE emp_stuff AS
  CURSOR c1 RETURN employees%ROWTYPE;  -- Declare cursor
END emp_stuff;
/
CREATE PACKAGE BODY emp_stuff AS
  CURSOR c1 RETURN employees%ROWTYPE IS
    SELECT * FROM employees WHERE salary > 2500;  -- Define cursor
END emp_stuff;
/
```

Example 11-8 ACCESSIBLE BY Clause

```
CREATE OR REPLACE PACKAGE helper
  AUTHID DEFINER
  ACCESSIBLE BY (api)
IS
  PROCEDURE h1;
  PROCEDURE h2;
END;
/

CREATE OR REPLACE PACKAGE BODY helper
IS
  PROCEDURE h1 IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE('Helper procedure h1');
  END;

  PROCEDURE h2 IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE('Helper procedure h2');
  END;
END;
/

CREATE OR REPLACE PACKAGE api
  AUTHID DEFINER
IS
  PROCEDURE p1;
  PROCEDURE p2;
END;
/

CREATE OR REPLACE PACKAGE BODY api
IS
  PROCEDURE p1 IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE('API procedure p1');
    helper.h1;
  END;

  PROCEDURE p2 IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE('API procedure p2');
    helper.h2;
  END;
END;
/
```

Invoke procedures in API package:

```
BEGIN
  api.p1;
  api.p2;
END;
/
```

Result:

```
API procedure p1
Helper procedure h1
```

```
API procedure p2
Helper procedure h2
```

Invoke a procedure in helper package:

```
BEGIN
  helper.h1;
END;
/
```

Result:

```
SQL> BEGIN
  2    helper.h1;
  3  END;
  4  /
  helper.h1;
  *
ERROR at line 2:
ORA-06550: line 2, column 3:
PLS-00904: insufficient privilege to access object HELPER
ORA-06550: line 2, column 3:
PL/SQL: Statement ignored
```

Package Example

Example 11-9 creates a table, `log`, and a package, `emp_admin`, and then invokes package subprograms from an anonymous block. The package has both specification and body.

The specification declares a public type, cursor, and exception, and three public subprograms. One public subprogram is overloaded (for information about overloaded subprograms, see "Overloaded Subprograms").

The body declares a private variable, defines the public cursor and subprograms that the specification declares, declares and defines a private function, and has an initialization part.

The initialization part (which runs only the first time the anonymous block references the package) inserts one row into the table `log` and initializes the private variable `number_hired` to zero. Every time the package procedure `hire_employee` is invoked, it updates the private variable `number_hired`.

Example 11-9 Creating emp_admin Package

```
-- Log to track changes (not part of package):

DROP TABLE log;
CREATE TABLE log (
  date_of_action  DATE,
  user_id         VARCHAR2(20),
  package_name    VARCHAR2(30)
);

-- Package specification:

CREATE OR REPLACE PACKAGE emp_admin AUTHID DEFINER AS
  -- Declare public type, cursor, and exception:
  TYPE EmpRecTyp IS RECORD (emp_id NUMBER, sal NUMBER);
  CURSOR desc_salary RETURN EmpRecTyp;
  invalid_salary EXCEPTION;
  -- Declare public subprograms:

  FUNCTION hire_employee (
    last_name       VARCHAR2,
    first_name      VARCHAR2,
    email           VARCHAR2,
    phone_number    VARCHAR2,
    job_id          VARCHAR2,
    salary          NUMBER,
    commission_pct  NUMBER,
    manager_id      NUMBER,
    department_id   NUMBER
  ) RETURN NUMBER;

  -- Overload preceding public subprogram:
  PROCEDURE fire_employee (emp_id NUMBER);
  PROCEDURE fire_employee (emp_email VARCHAR2);

  PROCEDURE raise_salary (emp_id NUMBER, amount NUMBER);
  FUNCTION nth_highest_salary (n NUMBER) RETURN EmpRecTyp;
END emp_admin;
/
-- Package body:

CREATE OR REPLACE PACKAGE BODY emp_admin AS
  number_hired  NUMBER;  -- private variable, visible only in this package

  -- Define cursor declared in package specification:

  CURSOR desc_salary RETURN EmpRecTyp IS
    SELECT employee_id, salary
    FROM employees
    ORDER BY salary DESC;

  -- Define subprograms declared in package specification:

  FUNCTION hire_employee (
    last_name       VARCHAR2,
    first_name      VARCHAR2,
    email           VARCHAR2,
    phone_number    VARCHAR2,
    job_id          VARCHAR2,
    salary          NUMBER,
    commission_pct  NUMBER,
    manager_id      NUMBER,
    department_id   NUMBER
  ) RETURN NUMBER
  IS
    new_emp_id NUMBER;
  BEGIN
    new_emp_id := employees_seq.NEXTVAL;
    INSERT INTO employees (
      employee_id,
      last_name,
      first_name,
      email,
      phone_number,
      hire_date,
      job_id,
      salary,
      commission_pct,
      manager_id,
      department_id
    )
    VALUES (
      new_emp_id,
      hire_employee.last_name,
      hire_employee.first_name,
      hire_employee.email,
      hire_employee.phone_number,
      SYSDATE,
      hire_employee.job_id,
      hire_employee.salary,
      hire_employee.commission_pct,
      hire_employee.manager_id,
      hire_employee.department_id
    );
    number_hired := number_hired + 1;
    DBMS_OUTPUT.PUT_LINE('The number of employees hired is '
                         || TO_CHAR(number_hired) );
    RETURN new_emp_id;
  END hire_employee;

  PROCEDURE fire_employee (emp_id NUMBER) IS
  BEGIN
    DELETE FROM employees WHERE employee_id = emp_id;
  END fire_employee;

  PROCEDURE fire_employee (emp_email VARCHAR2) IS
  BEGIN
    DELETE FROM employees WHERE email = emp_email;
  END fire_employee;

  -- Define private function, available only inside package:

  FUNCTION sal_ok (
    jobid VARCHAR2,
    sal NUMBER
  ) RETURN BOOLEAN
  IS
    min_sal NUMBER;
    max_sal NUMBER;
  BEGIN
    SELECT MIN(salary), MAX(salary)
    INTO min_sal, max_sal
    FROM employees
    WHERE job_id = jobid;

    RETURN (sal >= min_sal) AND (sal <= max_sal);
  END sal_ok;

  PROCEDURE raise_salary (
    emp_id NUMBER,
    amount NUMBER
  )
  IS
    sal NUMBER(8,2);
    jobid VARCHAR2(10);
  BEGIN
    SELECT job_id, salary INTO jobid, sal
    FROM employees
    WHERE employee_id = emp_id;

    IF sal_ok(jobid, sal + amount) THEN  -- Invoke private function
```

```
      UPDATE employees
      SET salary = salary + amount
      WHERE employee_id = emp_id;
    ELSE
      RAISE invalid_salary;
    END IF;
  EXCEPTION
    WHEN invalid_salary THEN
      DBMS_OUTPUT.PUT_LINE ('The salary is out of the specified range.');
  END raise_salary;

  FUNCTION nth_highest_salary (
    n NUMBER
  ) RETURN EmpRecTyp
  IS
    emp_rec  EmpRecTyp;
  BEGIN
    OPEN desc_salary;
    FOR i IN 1..n LOOP
      FETCH desc_salary INTO emp_rec;
    END LOOP;
    CLOSE desc_salary;
    RETURN emp_rec;
  END nth_highest_salary;

BEGIN  -- initialization part of package body
   INSERT INTO log (date_of_action, user_id, package_name)
   VALUES (SYSDATE, USER, 'EMP_ADMIN');
   number_hired := 0;
END emp_admin;
/
-- Invoke packages subprograms in anonymous block:

DECLARE
  new_emp_id NUMBER(6);
BEGIN
  new_emp_id := emp_admin.hire_employee (
    'Belden',
    'Enrique',
    'EBELDEN',
    '555.111.2222',
    'ST_CLERK',
    2500,
    .1,
    101,
    110
  );
  DBMS_OUTPUT.PUT_LINE ('The employee id is ' || TO_CHAR(new_emp_id));
  emp_admin.raise_salary (new_emp_id, 100);

  DBMS_OUTPUT.PUT_LINE (
    'The 10th highest salary is '||
    TO_CHAR (emp_admin.nth_highest_salary(10).sal) ||
             ', belonging to employee: ' ||
             TO_CHAR (emp_admin.nth_highest_salary(10).emp_id)
  );

  emp_admin.fire_employee(new_emp_id);
  -- You can also delete the newly added employee as follows:
  -- emp_admin.fire_employee('EBELDEN');
END;
/
```

Result is similar to:

```
The number of employees hired is 1
The employee id is 210
The 10th highest salary is 11500, belonging to employee: 168
```

How STANDARD Package Defines the PL/SQL Environment

A package named `STANDARD` defines the PL/SQL environment. The package specification declares public types, variables, exceptions, subprograms, which are available automatically to PL/SQL programs. For example, package `STANDARD` declares function `ABS`, which returns the absolute value of its argument, as follows:

```
FUNCTION ABS (n NUMBER) RETURN NUMBER;
```

The contents of package `STANDARD` are directly visible to applications. You need not qualify references to its contents by prefixing the package name. For example, you might invoke `ABS` from a database trigger, stored subprogram, Oracle tool, or 3GL application, as follows:

```
abs_diff := ABS(x - y);
```

If you declare your own version of `ABS`, your local declaration overrides the public declaration. You can still invoke the SQL function by specifying its full name:

```
abs_diff := STANDARD.ABS(x - y);
```

Most SQL functions are overloaded. For example, package `STANDARD` contains these declarations:

```
FUNCTION TO_CHAR (right DATE) RETURN VARCHAR2;
FUNCTION TO_CHAR (left NUMBER) RETURN VARCHAR2;
FUNCTION TO_CHAR (left DATE, right VARCHAR2) RETURN VARCHAR2;
FUNCTION TO_CHAR (left NUMBER, right VARCHAR2) RETURN VARCHAR2;
```

PL/SQL resolves an invocation of `TO_CHAR` by matching the number and data types of the formal and actual parameters.