PART V

PL/SQL Application Construction

This part of the book is where it all comes together. By now, you've learned the basics. You know about declaring and working with variables. You're an expert on error handling and loop construction. Now it's time to build an application—and you do that by constructing the building blocks, made up of procedures, functions, packages, and triggers, as described in Chapter 17 through Chapter 19. Chapter 20 discusses managing your PL/SQL code base, including testing and debugging programs and managing dependencies; it also provides an overview of the edition-based redefinition capability introduced in Oracle Database 11*g* Release 2. Chapter 21 focuses on how you can use a variety of tools and techniques to get the best performance out of your PL/SQL programs. Chapter 22 describes I/O techniques for PL/SQL, from DBMS_OUTPUT (sending output to the screen) and UTL_FILE (reading and writing files) to UTL_MAIL (sending mail) and UTL_HTTP (retrieving data from a web page).

CHAPTER 17

Procedures, Functions, and Parameters

Earlier parts of this book have explored in detail all of the components of the PL/SQL language: cursors, exceptions, loops, variables, and so on. While you certainly need to know about these components when you write applications using PL/SQL, putting the pieces together to create well-structured, easily understood, and smoothly maintainable programs is even more important.

Few of our tasks are straightforward. Few solutions can be glimpsed in an instant and immediately put to paper or keyboard. The systems we build are usually large and complex, with many interacting and sometimes conflicting components. Furthermore, as users deserve, demand, and receive applications that are easier to use and vastly more powerful than their predecessors, the inner world of those applications becomes correspondingly more complicated.

One of the biggest challenges in our profession today is finding ways to reduce the complexity of our environment. When faced with a massive problem to solve, the mind is likely to recoil in horror. Where do I start? How can I possibly find a way through that jungle of requirements and features?

A human being is not a massively parallel computer. Even the brightest of our bunch have trouble keeping track of more than seven tasks (plus or minus two) at one time. We need to break down huge, intimidating projects into smaller, more manageable components, and then further decompose those components into individual programs with an understandable scope. We can then figure out how to build and test those programs, after which we can construct a complete application from these building blocks.

Whether you use *top-down design* (a.k.a. *stepwise refinement*, which is explored in detail in the section "Local or Nested Modules" on page 619) or some other methodology, there is absolutely no doubt that you will find your way to a high-quality and easily maintainable application by modularizing your code into procedures, functions, and object types.

Modular Code

Modularization is the process by which you break up large blocks of code into smaller pieces (modules) that can be called by other modules. Modularization of code is analogous to normalization of data, with many of the same benefits and a few additional advantages. With modularization, your code becomes:

More reusable
: When you break up a large program (or entire application) into individual components that "plug and play" together, you will usually find that many modules are used by one or more other programs in your current application. Designed properly, these utility programs could even be of use in other applications!

More manageable
: Which would you rather debug: a 1,000-line program or five individual 200-line programs that call each other as needed? Our minds work better when we can focus on smaller tasks. With modularized code, you can also test and debug on a per-program scale (called *unit testing*) before combining individual modules for a more complicated integration test.

More readable
: Modules have names, and names describe behavior. The more you move or hide your code behind a programmatic interface, the easier it is to read and understand what that program is doing. Modularization helps you focus on the big picture rather than on the individual executable statements. You might even end up with that most elusive kind of software: a self-documenting program.

More reliable
: The code you produce will have fewer errors. The errors you do find will be easier to fix because they will be isolated within a module. In addition, your code will be easier to maintain because there is less of it and it is more readable.

Once you become proficient with the different iterative, conditional, and cursor constructs of the PL/SQL language (the IF statement, loops, etc.), you are ready to write programs. You will not really be ready to build an application, however, until you understand how to create and combine PL/SQL modules.

PL/SQL offers the following types of structure that modularize your code in different ways:

Procedure
: A program that performs one or more actions and is called as an executable PL/SQL statement. You can pass information into and out of a procedure through its parameter list.

Function
: A program that returns data through its RETURN clause, and is used just like a PL/SQL expression. You can pass information into a function through its parameter list. You can also pass information out via the parameter list, but this is generally considered a bad practice.

Database trigger
: A set of commands that are triggered to execute (e.g., log in, modify a row in a table, execute a DDL statement) when an event occurs in the database.

Package
: A named collection of procedures, functions, types, and variables. A package is not really a module (it's more of a metamodule), but it is so closely related that I mention it here.

Object type or instance of an object type
: Oracle's version of (or attempt to emulate) an object-oriented class. Object types encapsulate state and behavior, combining data (like a relational table) with rules (procedures and functions that operate on that data).

Packages are discussed in Chapter 18; database triggers are explored in Chapter 19. You can read more about object types in Chapter 26. This chapter focuses on how to build procedures and functions, and how to design the parameter lists that are an integral part of well-designed modules.

I use the term *module* to mean either a function or a procedure. As is the case with many other programming languages, modules can call other named modules. You can pass information into and out of modules with parameters. Finally, the modular structure of PL/SQL also integrates tightly with exception handlers to provide all-encompassing error-checking techniques (see Chapter 6).

This chapter explores how to define procedures and functions, and then dives into the details of setting up parameter lists for these programs. I also examine some of the more "exotic" aspects of program construction, including local modules, overloading, forward referencing, deterministic functions, and table functions.

Procedures

A *procedure* is a module that performs one or more actions. Because a procedure call is a standalone executable statement in PL/SQL, a PL/SQL block could consist of nothing more than a single call to a procedure. Procedures are key building blocks of modular code, allowing you to both consolidate and reuse your program logic.

The general format of a PL/SQL procedure is as follows:

```
PROCEDURE [schema.]name[( parameter[, parameter...] ) ]
   [AUTHID DEFINER | CURRENT_USER ]
   [ACCESSIBLE BY (program_unit_list)]
IS
   [declarations]

BEGIN
   executable statements

[ EXCEPTION
     exception handlers]

 END [name];
```

where:

schema
: Is the (optional) name of the schema that will own this procedure. The default is the current user. If different from the current user, that user will need the CREATE ANY PROCEDURE privilege to create a procedure in another schema.

name
: Is the name of the procedure.

(parameters[, parameter...])
: Is an optional list of parameters that you define to both pass information to the procedures and send information out of the procedure back to the calling program.

AUTHID clause
: Determines whether the procedure will execute with the privileges of the definer (owner) of the procedure or the current user. The former (the default) is known as the *definer rights model*, the latter as the *invoker rights model*. These models are described in detail in Chapter 24.

declarations
: Declare any local identifiers for the procedure. If you do not have any declarations, there will be no statements between the IS and BEGIN statements.

ACCESSIBLE BY clause (Oracle Database 12c)
: Restricts access to this procedure to the program units listed inside the parentheses. This feature is explored in Chapter 24.

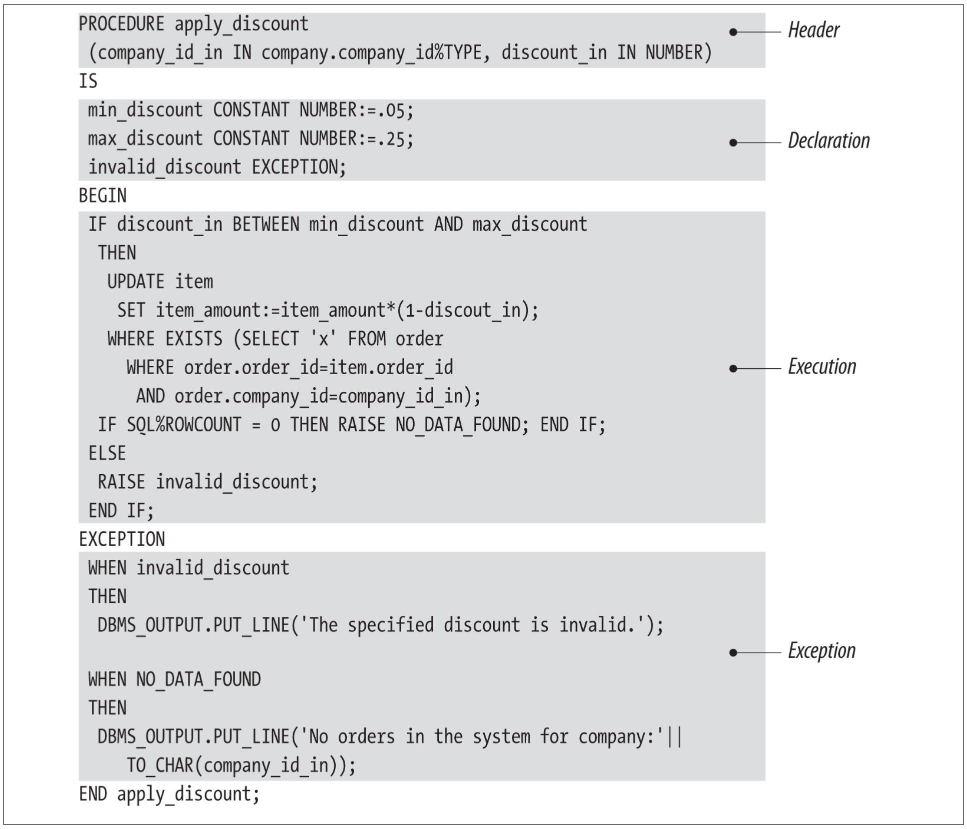
executable statements

Are the statements that the procedure executes when it is called. You must have at least one executable statement after the BEGIN keyword and before the END or EXCEPTION keyword.

exception handlers

Specify any exception handlers for the procedure. If you do not explicitly handle any exceptions, then you can leave out the EXCEPTION keyword and simply terminate the execution section with the END keyword.

Figure 17-1 shows the apply_discount procedure, which contains all four sections of the named PL/SQL block as well as a parameter list.



```
PROCEDURE apply_discount
 (company_id_in IN company.company_id%TYPE, discount_in IN NUMBER)
IS
 min_discount CONSTANT NUMBER:=.05;
 max_discount CONSTANT NUMBER:=.25;
 invalid_discount EXCEPTION;
BEGIN
 IF discount_in BETWEEN min_discount AND max_discount
  THEN
   UPDATE item
    SET item_amount:=item_amount*(1-discout_in);
   WHERE EXISTS (SELECT 'x' FROM order
     WHERE order.order_id=item.order_id
      AND order.company_id=company_id_in);
  IF SQL%ROWCOUNT = 0 THEN RAISE NO_DATA_FOUND; END IF;
 ELSE
  RAISE invalid_discount;
 END IF;
EXCEPTION
 WHEN invalid_discount
 THEN
  DBMS_OUTPUT.PUT_LINE('The specified discount is invalid.');

 WHEN NO_DATA_FOUND
 THEN
  DBMS_OUTPUT.PUT_LINE('No orders in the system for company:'||
     TO_CHAR(company_id_in));
END apply_discount;
```

Figure 17-1. The apply_discount procedure

Calling a Procedure

A procedure is called as an executable PL/SQL statement. In other words, a call to a procedure must end with a semicolon (;) and be executed before or after other SQL or PL/SQL statements (if they exist) in the execution section of a PL/SQL block.

The following executable statement runs the apply_discount procedure:

```
BEGIN
   apply_discount( new_company_id, 0.15 );  -- 15% discount
END;
```

If the procedure does not have any parameters, then you may call the procedure with or without parentheses, as shown here:

```
display_store_summary;
display_store_summary();
```

The Procedure Header

The portion of the procedure definition that comes before the IS keyword is called the *procedure header* or *signature*. The header provides all the information a programmer needs to call that procedure, namely:

- The procedure name
- The AUTHID clause, if any
- The parameter list, if any
- The accessible-by list, if any (new to Oracle Database 12*c*)

Ideally, a programmer should only need to see the header of the procedure in order to understand what it does and how it is to be called.

The header for the apply_discount procedure mentioned in the previous section is:

```
PROCEDURE apply_discount (
    company_id_in IN company.company_id%TYPE
   , discount_in IN NUMBER
)
```

It consists of the module type, the name, and a list of two parameters.

The Procedure Body

The body of the procedure is the code required to implement that procedure; it consists of the declaration, execution, and exception sections of the procedure. Everything after the IS keyword in the procedure makes up that procedure's body. The exception and declaration sections are optional. If you have no exception handlers, leave off the EXCEPTION keyword and simply enter the END statement to terminate the procedure.

If you have no declarations, the BEGIN statement simply follows immediately after the IS keyword.

You must supply at least one executable statement in a procedure. That is generally not a problem; instead, watch out for execution sections that become extremely long and hard to manage. You should work hard to keep the execution section compact and readable. See later sections in this chapter, especially "Improving readability" on page 621, for more specific guidance on this topic.

The END Label

You can append the name of the procedure directly after the END keyword when you complete your procedure, as shown here:

```
PROCEDURE display_stores (region_in IN VARCHAR2) IS
BEGIN
   ...
END display_stores;
```

This name serves as a label that explicitly links the end of the program with its beginning. You should, as a matter of habit, use an END label. It is especially important to do so when you have a procedure that spans more than a single page, or is one in a series of procedures and functions in a package body.

The RETURN Statement

The RETURN statement is generally associated with a function because it is required to RETURN a value from a function (or else raise an exception). Interestingly, PL/SQL also allows you to use a RETURN statement in a procedure. The procedure version of the RETURN does not take an expression; it therefore cannot pass a value back to the calling program unit. The RETURN simply halts execution of the procedure and returns control to the calling code.

You do not see this usage of RETURN very often, and for good reason. Use of the RETURN in a procedure usually leads to unstructured code because there will then be at least two paths out of the procedure, making execution flow harder to understand and maintain. Avoid using both RETURN and GOTO to bypass proper control structures and process flow in your program units.

Functions

A *function* is a module that returns data through its RETURN clause, rather than in an OUT or IN OUT argument. Unlike a procedure call, which is a standalone executable statement, a call to a function can exist only as part of an executable statement, such as an element in an expression or the value assigned as the default in a declaration of a variable.

Because a function returns a value, it is said to have a datatype. A function can be used in place of an expression in a PL/SQL statement having the same datatype as the function.

Functions are particularly important constructs for building modular code. For example, every single business rule or formula in your application should be placed inside a function. Rather than writing the same queries over and over again ("Get the name of the employee from his ID", "Get the latest order row from the order table for this company ID", and so on), put each query inside its own function, and call that function in multiple places. The result is code that is more easily debugged, optimized, and maintained.

Some programmers prefer to rely less on functions, and more on procedures that return status information through the parameter list. If you are one of these programmers, make sure that your business rules, formulas, and single-row queries are tucked away into your procedures!

An application short on function definition and usage is likely to be difficult to maintain and enhance over time.

Structure of a Function

The structure of a function is the same as that of a procedure, except that the function also has a RETURN clause. The format of a function is as follows:

```
FUNCTION [schema.]name[( parameter[, parameter...] ) ]
   RETURN return_datatype
   [AUTHID DEFINER | CURRENT_USER]
   [DETERMINISTIC]
   [PARALLEL_ENABLE ...]
   [PIPELINED]
   [RESULT_CACHE ...]
   [ACCESSIBLE BY (program_unit_list)
   [AGGREGATE ...]
   [EXTERNAL ...]
IS
   [declaration statements]

BEGIN
   executable statements

[EXCEPTION
   exception handler statements]

END [name];
```

where:

schema
: Is the (optional) name of the schema that will own this function. The default is the current user. If different from the current user, that user will need the CREATE ANY PROCEDURE privilege to create a function in another schema.

name
: Is the name of the function.

(parameters[, parameter...])
: Is an optional list of parameters that you define to both pass information into the function and send information out of the function back to the calling program.

return_datatype
: Specifies the datatype of the value returned by the function. This is required in the function header and is explained in more detail in the next section.

AUTHID clause
: Determines whether the function will execute with the privileges of the definer (owner) of the procedure or of the current user. The former (the default) is known as the *definer rights model*, the latter as the *invoker rights model*.

DETERMINISTIC clause
: Defines this function to be *deterministic*, which means that the value returned by the function is determined completely by the argument values. If you include this clause, you will be able to use the function in a function-based index and the SQL engine may be able to optimize execution of the function when it is called inside queries. See "Deterministic Functions" on page 647 for more details.

PARALLEL_ENABLE clause
: Is an optimization hint that enables the function to be executed in parallel when called from within a SELECT statement.

PIPELINED clause
: Specifies that the results of this table function should be returned iteratively via the PIPE ROW command.

RESULT_CACHE clause
: Specifies that the input values and result of this function should be stored in the new function result cache. This feature, introduced in Oracle Database 11*g*, is explored in detail in Chapter 21.

ACCESSIBLE BY clause (Oracle Database 12c)
: Restricts access to this function to the program units listed inside the parentheses. This feature is explored in Chapter 24.

AGGREGATE clause
: Is used when you are defining your own aggregate function.

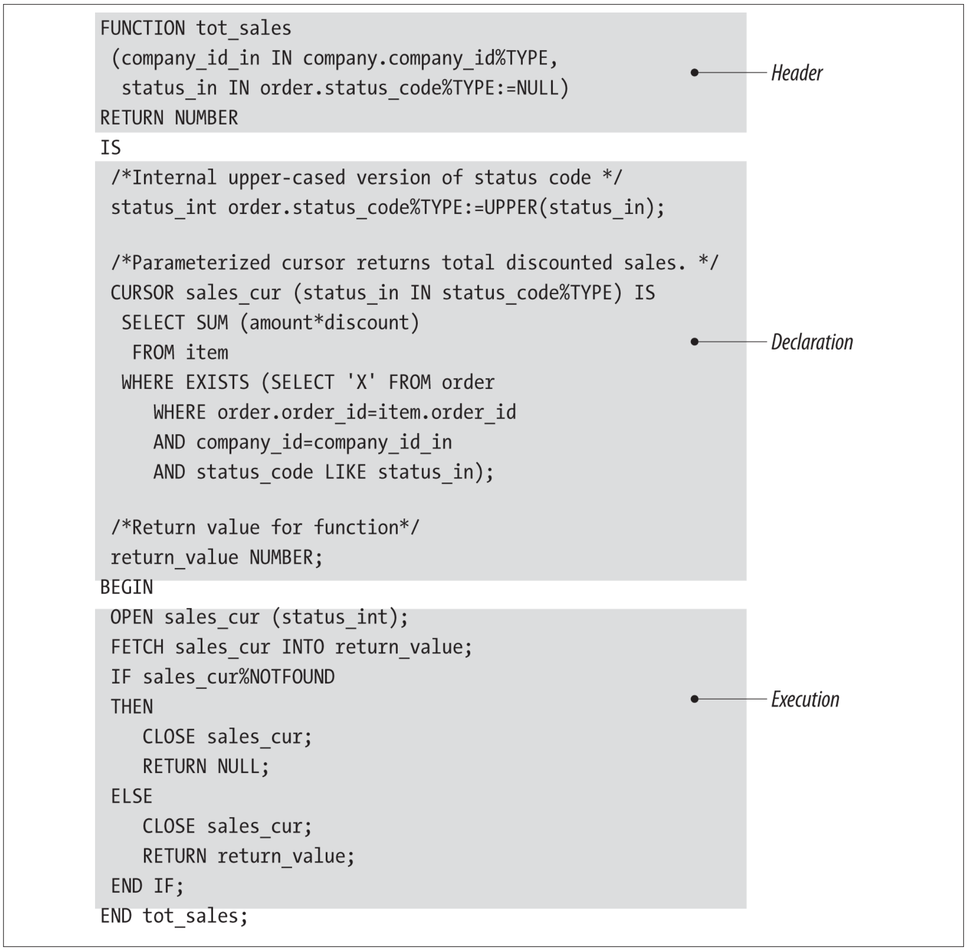
EXTERNAL clause
: Is used to define the function as implemented "externally"—i.e., as C code.

declaration statements
: Declare any local identifiers for the function. If you do not have any declarations, there will be no statements between the IS and BEGIN statements.

executable statements
: Are the statements the function executes when it is called. You must have at least one executable statement after the BEGIN keyword and before the END or EXCEPTION keyword.

exception handler statements
: Specify any exception handlers for the function. If you do not explicitly handle any exceptions, then you can leave out the EXCEPTION keyword and simply terminate the execution section with the END keyword.

Figure 17-2 illustrates the PL/SQL function and its different sections. Notice that the total_sales function does not have an exception section.



```
FUNCTION tot_sales
 (company_id_in IN company.company_id%TYPE,
  status_in IN order.status_code%TYPE:=NULL)
RETURN NUMBER
IS
 /*Internal upper-cased version of status code */
 status_int order.status_code%TYPE:=UPPER(status_in);

 /*Parameterized cursor returns total discounted sales. */
 CURSOR sales_cur (status_in IN status_code%TYPE) IS
  SELECT SUM (amount*discount)
   FROM item
  WHERE EXISTS (SELECT 'X' FROM order
     WHERE order.order_id=item.order_id
     AND company_id=company_id_in
     AND status_code LIKE status_in);

 /*Return value for function*/
 return_value NUMBER;
BEGIN
 OPEN sales_cur (status_int);
 FETCH sales_cur INTO return_value;
 IF sales_cur%NOTFOUND
 THEN
    CLOSE sales_cur;
    RETURN NULL;
 ELSE
    CLOSE sales_cur;
    RETURN return_value;
 END IF;
END tot_sales;
```

Figure 17-2. The tot_sales function

The RETURN Datatype

A PL/SQL function can return virtually any kind of data known to PL/SQL, from scalars (single, primitive values like dates and strings) to complex structures such as collections, object types, cursor variables, and LOBs.

Here are some examples of RETURN clauses in functions:

- Return a string from a standalone function:

```
FUNCTION favorite_nickname (
   name_in IN VARCHAR2) RETURN VARCHAR2
IS
BEGIN
```

```
      ...
    END;
```

- Return a number (age of a pet) from an object type member function:

  ```
  TYPE pet_t IS OBJECT (
     tag_no              INTEGER,
     NAME                VARCHAR2 (60),
     breed               VARCHAR2(100),
     dob DATE,
     MEMBER FUNCTION age  RETURN NUMBER
  )
  ```

- Return a record with the same structure as the books table:

  ```
  PACKAGE book_info
  IS
     FUNCTION onerow (isbn_in IN books.isbn%TYPE)
       RETURN books%ROWTYPE;
  ...
  ```

- Return a cursor variable with the specified REF CURSOR type (based on a record type):

  ```
  PACKAGE book_info
  IS
     TYPE overdue_rt IS RECORD (
        isbn books.isbn%TYPE,
        days_overdue PLS_INTEGER);

     TYPE overdue_rct IS REF CURSOR RETURN overdue_rt;


     FUNCTION overdue_info (username_in IN lib_users.username%TYPE)
       RETURN overdue_rct;

  ...
  ```

The END Label

You can append the name of the function directly after the END keyword when you complete your function, as shown here:

```
FUNCTION total_sales (company_in IN INTEGER) RETURN NUMBER
IS
BEGIN
   ...
END total_sales;
```

This name serves as a label that explicitly links the end of the program with its beginning. You should, as a matter of habit, use an END label. It is especially important to do so when you have a function that spans more than a single page or that is one in a series of functions and procedures in a package body.

Calling a Function

A function is called as part of an executable PL/SQL statement wherever an expression can be used. The following examples illustrate how the various functions defined in the section "The RETURN Datatype" on page 601 can be invoked:

- Assign the default value of a variable with a function call:

```
DECLARE
   v_nickname VARCHAR2(100) :=
      favorite_nickname ('Steven');
```

- Use a member function for the pet object type in a conditional expression:

```
DECLARE
   my_parrot pet_t :=
      pet_t (1001, 'Mercury', 'African Grey',
             TO_DATE ('09/23/1996', 'MM/DD/YYYY'));
BEGIN
   IF my_parrot.age () < INTERVAL '50' YEAR
    THEN
      DBMS_OUTPUT.PUT_LINE ('Still a youngster!');
   END IF;
```

- Retrieve a single row of book information directly into a record:

```
DECLARE
   my_first_book books%ROWTYPE;
BEGIN
   my_first_book := book_info.onerow ('1-56592-335-9');
   ...
```

- Call a user-defined PL/SQL function from within a query:

```
DECLARE
  l_name employees.last_name%TYPE;
BEGIN

  SELECT last_name INTO l_name
    FROM employees
   WHERE employee_id = hr_info_pkg.employee_of_the_month ('FEBRUARY');
   ...
```

- Call a function of your own making from within a CREATE VIEW statement, utilizing a CURSOR expression to pass a result set as an argument to that function:

```
VIEW young_managers
AS
  SELECT managers.employee_id AS manager_employee_id
    FROM employees managers
    WHERE most_reports_before_manager
```

```
            (
              CURSOR ( SELECT reports.hire_date
                         FROM employees reports
                         WHERE reports.manager_id = managers.employee_id
                      ),
              managers.hire_date
            ) = 1;
```

With PL/SQL, in contrast to some other programming languages, you cannot simply ignore the return value of a function if you don't need it. For example, this function call:

```
BEGIN
     favorite_nickname('Steven');
END;
```

will raise the error *PLS-00221: 'FAVORITE_NICKNAME' is not a procedure or is undefined*. You may not use a function as if it were a procedure.

Functions Without Parameters

If a function has no parameters, the function call can be written with or without parentheses. The following code illustrates this with a call to a method named age of the pet_t object type:

```
IF my_parrot.age < INTERVAL '50' YEAR -- 9i INTERVAL type
IF my_parrot.age() < INTERVAL '50' YEAR
```

The Function Header

The portion of the function definition that comes before the IS keyword is called the *function header* or *signature*. The header provides all the information a programmer needs to call that function, namely:

- The function name
- Modifiers to the definition and behavior of the function (e.g., is it deterministic? Does it run in parallel execution? Is it pipelined?)
- The parameter list, if any
- The RETURN datatype

Ideally, a programmer should need to look only at the header of the function in order to understand what it does and how it is to be called.

The header for the total_sales function discussed earlier is:

```
FUNCTION total_sales
   (company_id_in IN company.company_id%TYPE,
    status_in IN order.status_code%TYPE := NULL)
RETURN NUMBER
```

It consists of the module type, the name, a list of two parameters, and a RETURN datatype of NUMBER. This means that any PL/SQL statement or expression that references a numeric value can make a call to total_sales to obtain that value. Here is one such statement:

```
DECLARE
   v_sales NUMBER;
BEGIN
   v_sales := total_sales (1505, 'ACTIVE');
   ...
END;
```

The Function Body

The body of the function is the code required to implement the function. It consists of the declaration, execution, and exception sections of the function. Everything after the IS keyword in the function makes up that function's body.

Once again, the declaration and exception sections are optional. If you have no exception handlers, simply leave off the EXCEPTION keyword and enter the END statement to terminate the function. If you have no declarations, the BEGIN statement simply follows immediately after the IS keyword.

A function's execution section should have a RETURN statement in it. This is not necessary for the function to compile but if your function finishes executing without executing a RETURN statement, Oracle will raise the following error (a sure sign of a very poorly designed function):

```
ORA-06503: PL/SQL: Function returned without value
```

This error will *not* be raised if the function propagates an exception of its own unhandled out of the function.

The RETURN Statement

A function must have at least one RETURN statement in its execution section. It can have more than one RETURN, but only one is executed each time the function is called. The RETURN statement that is executed by the function determines the value that is returned by that function. When a RETURN statement is processed, the function terminates immediately and returns control to the calling PL/SQL block.

The RETURN clause in the header of the function is different from the RETURN statement in the execution section of the body. While the RETURN clause indicates the datatype of the return or result value of the function, the RETURN statement specifies the actual value that is returned. You have to specify the RETURN datatype in the header, but then also include at least one RETURN statement in the function. The datatype indicated in the RETURN clause in the header must be compatible with the datatype of the returned expression in the RETURN statement.

RETURN any valid expression

The RETURN statement can return any expression compatible with the datatype indicated in the RETURN clause. This expression can be composed of calls to other functions, complex calculations, and even data conversions. All of the following usages of RETURN are valid:

```
RETURN 'buy me lunch';
RETURN POWER (max_salary, 5);
RETURN (100 - pct_of_total_salary (employee_id));
RETURN TO_DATE ('01' || earliest_month || initial_year, 'DDMMYY');
```

You can also return complex data structures such as object type instances, collections, and records.

An expression in the RETURN statement is evaluated when the RETURN is executed. When control is passed back to the calling block, the result of the evaluated expression is passed along, too.

Multiple RETURNs

In the total_sales function shown in Figure 17-2, I used two different RETURN statements to handle different situations in the function, which can be described as follows:

> If I cannot obtain sales information from the cursor, I return NULL (which is different from zero). If I do get a value from the cursor, I return it to the calling program. In both of these cases, the RETURN statement passes back a value: in one case, the NULL value, and in the other, the return_value variable.

While it is certainly possible to have more than one RETURN statement in the execution section of a function, you are generally better off having just one: the last line in your execution section. The next section explains this recommendation.

RETURN as the last executable statement

Generally, the best way to make sure that your function always returns a value is to make the last executable statement your RETURN statement. Declare a variable named return_value (which clearly indicates that it will contain the return value for the function), write all the code to come up with that value, and then, at the very end of the function, RETURN the return_value, as shown here:

```
FUNCTION do_it_all (parameter_list) RETURN NUMBER IS
   return_value NUMBER;
BEGIN
   ... lots of executable statements ...
```

```
      RETURN return_value;
   END;
```

Here is a rewrite of the logic in Figure 17-2 to fix the problem of multiple RETURN statements:

```
   OPEN sales_cur;
   IF sales_cur%NOTFOUND
   THEN
      return_value:= NULL;
   END IF;
   CLOSE sales_cur;
   RETURN return_value;
```

Beware of exceptions, though. An exception that gets raised might "jump" over your last statement straight into the exception handler. If your exception handler does not then have a RETURN statement, you will get an *ORA-06503: Function returned without value* error, regardless of how you handled the actual exception (unless you RAISE another).

Parameters

Procedures and functions can both use *parameters* to pass information back and forth between the module and the calling PL/SQL block.

The parameters of a module, part of its header or signature, are at least as important as the code that implements the module (the module's body). In fact, the header of the program is sometimes described as a contract between the author of the program and its users. Sure, you have to make certain that your module fulfills its promise. But the whole point of creating a module is that it can be called, ideally by more than one other module. If the parameter list is confusing or badly designed, it will be very difficult for other programmers to use the module, and the result is that few will bother. And it doesn't matter how well you implemented a program if no one uses it.

Many developers do not give enough attention to a module's set of parameters. Considerations regarding parameters include:

Number of parameters
: Too few parameters can limit the reusability of your program; with too many parameters, no one will want to reuse your program. Certainly, the number of parameters is largely determined by program requirements, but there are different ways to define parameters (such as bundling multiple parameters in a single record).

Types of parameters
: Should you use read-only, write-only, or read/write parameters?

Names of parameters
: How should you name your parameters so that their purpose in the module is properly and easily understood?

Default values for parameters
: How do you set defaults? When should a parameter be given a default, and when should the programmer be forced to enter a value?

PL/SQL offers many different features to help you design parameters effectively. This section covers all elements of parameter definition.

Defining Parameters

Formal parameters are defined in the parameter list of the program. A parameter definition parallels closely the syntax for declaring variables in the declaration section of a PL/SQL block. There are two important distinctions: first, a parameter has a passing mode, while a variable declaration does not; and second, a parameter declaration must be unconstrained.

A *constrained declaration* is one that constrains or limits the kind of value that can be assigned to a variable declared with that datatype. An *unconstrained declaration* is one that does not limit values in this way. The following declaration of the variable company_name constrains the variable to 60 characters:

```
DECLARE
   company_name VARCHAR2(60);
```

When you declare a parameter, however, you must leave out the constraining part of the declaration:

```
PROCEDURE display_company (company_name IN VARCHAR2) IS ...
```

Actual and Formal Parameters

We need to distinguish between two different kinds of parameters: actual and formal parameters. The *formal parameters* are the names that are declared in the parameter list of the header of a module. The *actual parameters* are the variables or expressions placed in the parameter list of the actual call to the module.

Let's examine the differences between formal and actual parameters using the example of total_sales. Here, again, is the total_sales header:

```
FUNCTION total_sales
   (company_id_in IN company.company_id%TYPE,
    status_in IN order.status_code%TYPE DEFAULT NULL)
RETURN std_types.dollar_amount;
```

The formal parameters of total_sales are:

company_id_in
: The primary key of the company

status_in
: The status of the orders to be included in the sales calculation

These formal parameters do not exist outside of the module. You can think of them as placeholders for real or actual argument values that are passed into the module when it is used in a program.

When I want to call total_sales, I must provide two arguments, which could be variables, constants, or literals (they *must* be variables if the parameter mode is OUT or IN OUT). In the following example, the company_id variable contains the primary key pointing to a company record. In the first three calls to total_sales, a different, hardcoded status is passed to the function. The last call to total_sales does not specify a status; in this case, the function assigns the default value (provided in the function header) to the status_in parameter:

```
new_sales      := total_sales (company_id, 'N');
paid_sales     := total_sales (company_id, 'P');
shipped_sales  := total_sales (company_id, 'S');
all_sales      := total_sales (company_id);
```

When total_sales is called, all the actual parameters are evaluated. The results of the evaluations are then assigned to the formal parameters inside the function to which they correspond (note that this is true only for IN and IN OUT parameters; parameters of mode OUT are not copied in).

The formal parameter and the actual parameter that corresponds to it (when called) must be of the same or compatible datatypes. PL/SQL will perform datatype conversions for you in many situations. Generally, however, you are better off avoiding all implicit datatype conversions. Use a formal conversion function like TO_CHAR (see "Numbers" on page 177) or TO_DATE (see Chapter 10), so that you know exactly what kind of data you are passing into your modules.

Parameter Modes

When you define a parameter, you can also specify the way in which it can be used. There are three different modes of parameters, described in the following table.

Mode	Description	Parameter usage
IN	Read-only	The value of the actual parameter can be referenced inside the module, but the parameter cannot be changed. If you do not specify the parameter mode, then it is considered an IN parameter.
OUT	Write-only (sort of)	The module can assign a value to the parameter, but the parameter's value cannot be read. Well, that's the official version, and a very reasonable definition of an OUT mode parameter. In reality, Oracle *does* let you read the value of an OUT parameter inside the subprogram.
IN OUT	Read/write	The module can both reference (read) and modify (write) the parameter.

The mode determines how the program can use and manipulate the value assigned to the formal parameter. You specify the mode of the parameter immediately after the parameter name and before the parameter's datatype and optional default value. The following procedure header uses all three parameter modes:

```
PROCEDURE predict_activity
   (last_date_in IN DATE,
    task_desc_inout IN OUT VARCHAR2,
    next_date_out OUT DATE)
```

The predict_activity procedure takes in two pieces of information: the date of the last activity and a description of the activity. It then returns or sends out two pieces of information: a possibly modified task description and the date of the next activity. Because the task_desc_inout parameter is of mode IN OUT, the program can both read the value of the argument and change the value of that argument.

Let's look at each of these parameter modes in detail.

IN mode

An IN parameter allows you to pass values into the module but will not pass anything out of the module and back to the calling PL/SQL block. In other words, for the purposes of the program, IN parameters function like constants. Just like a constant, the value of a formal IN parameter cannot be changed within the program. You cannot assign values to the IN parameter, or modify its value in any other way, without receiving a compilation error.

IN is the default mode; if you do not specify a parameter mode, the parameter is automatically considered IN. I recommend, however, that you always specify a parameter mode so that your intended use of the parameter is documented explicitly in the code itself.

IN parameters can be given default values in the program header (see the section "Default Values" on page 618).

The actual value for an IN parameter can be a variable, a named constant, a literal, or a complex expression. All of the following calls to display_title are valid:

```
/* File on web: display_title.sql */
DECLARE
   happy_title CONSTANT VARCHAR2(30) := 'HAPPY BIRTHDAY';
   changing_title VARCHAR2(30) := 'Happy Anniversary';
   spc CONSTANT VARCHAR2(1) := CHR(32); -- ASCII code for a single space
BEGIN
   display_title ('Happy Birthday');             -- a literal
   display_title (happy_title);                  -- a constant

   changing_title := happy_title;
   display_title (changing_title);               -- a variable
   display_title ('Happy' || spc || 'Birthday'); -- an expression
   display_title (INITCAP (happy_title));        -- another expression
END;
```

What if you want to transfer data out of your program? For that, you will need an OUT or an IN OUT parameter—or perhaps you might want to consider changing the procedure to a function.

OUT mode

An OUT parameter is the opposite of an IN parameter, but perhaps you already had that figured out. You can use an OUT parameter to pass a value back from the program to the calling PL/SQL block. An OUT parameter is like the return value for a function, but it appears in the parameter list, and you can have as many as you like (disclosure: PL/SQL allows a maximum of 64,000 parameters, but in practical terms, that is no limit at all).

Inside the program, an OUT parameter acts like a variable that has not been initialized. In fact, the OUT parameter has no value at all until the program terminates successfully (unless you have requested use of the NOCOPY hint, which is explored in detail in Chapter 21). During the execution of the program, any assignments to an OUT parameter are actually made to an internal copy of the OUT parameter. When the program terminates successfully and returns control to the calling block, the value in that local copy is transferred to the actual OUT parameter. That value is then available in the calling PL/SQL block.

There are several consequences of these rules concerning OUT parameters:

- You also cannot provide a default value to an OUT parameter. You can assign a value to an OUT parameter only inside the body of the module.
- Any assignments made to OUT parameters are rolled back when an exception is raised in the program. Because the value for an OUT parameter is not actually assigned until a program completes successfully, any intermediate assignments are therefore ignored. Unless an exception handler traps the exception and then assigns a value to the OUT parameter, no assignment is made to that parameter. The variable will retain the same value it had before the program was called.
- An actual parameter corresponding to an OUT formal parameter cannot be a constant, literal, or expression. Oracle has to be able to put a value *into* what you pass for the OUT parameter.

As noted in the table earlier, Oracle *does* allow you to read the value of an OUT parameter in a subprogram. That value is always initially NULL, but once you assign it a value in the subprogram, you can “see” that value, as demonstrated in the following script:

```
SQL> set serveroutput on
SQL> CREATE OR REPLACE PROCEDURE read_out (n OUT NUMBER)
  2  IS
```

```
BEGIN
   DBMS_OUTPUT.put_line ('n initial=' || n);
   n := 1;
   DBMS_OUTPUT.put_line ('n after assignment=' || n);
END;
/

Procedure created.

SQL> DECLARE
   n   NUMBER;
BEGIN
   read_out (n);
END;
/
n initial=
n after assignment=1
```

IN OUT mode

With an IN OUT parameter, you can pass values into the program and return a value back to the calling program (either the original, unchanged value or a new value set within the program). The IN OUT parameter shares two restrictions with the OUT parameter:

- An IN OUT parameter cannot be a constant, literal, or expression.
- An IN OUT actual parameter or argument must be a variable. It cannot be a constant, literal, or expression, because these formats do not provide a receptacle in which PL/SQL can place the outgoing value.

Beyond these restrictions, no other restrictions apply.

You can use an IN OUT parameter on both sides of an assignment because it functions like an initialized, rather than uninitialized, variable. PL/SQL does not lose the value of an IN OUT parameter when it begins execution of the program. Instead, it uses that value as necessary within the program.

The combine_and_format_names procedure shown here combines first and last names into a full name in the format specified ("LAST, FIRST" or "FIRST LAST"). I need the incoming names for the combine action, and I will uppercase the first and last names for future use in the program (thereby enforcing the application standard of all-uppercase for names of people and things). This program uses all three parameter modes:

```
PROCEDURE combine_and_format_names
   (first_name_inout IN OUT VARCHAR2,
    last_name_inout IN OUT VARCHAR2,
    full_name_out OUT VARCHAR2,
    name_format_in IN VARCHAR2 := 'LAST, FIRST')
IS
BEGIN
   /* Uppercase the first and last names. */
   first_name_inout := UPPER (first_name_inout);
   last_name_inout := UPPER (last_name_inout);

   /* Combine the names as directed by the name format string. */
   IF name_format_in = 'LAST, FIRST'
   THEN
      full_name_out := last_name_inout || ', ' || first_name_inout;

   ELSIF name_format_in = 'FIRST LAST'
   THEN
      full_name_out := first_name_inout || ' ' || last_name_inout;
   END IF;
END combine_and_format_names;
```

The first name and last name parameters must be IN OUT. full_name_out is just an OUT parameter because I create the full name from its parts. If the actual parameter used to receive the full name has a value going into the procedure, I certainly don't want to use it! Finally, the name_format_in parameter is a mere IN parameter because it is used to determine how to format the full name, but is not changed or changeable in any way.

Each parameter mode has its own characteristics and purpose. You should choose carefully which mode to apply to your parameters so that they are used properly within the module.

> You should define formal parameters with OUT or IN OUT modes only in procedures. Functions should return all their information only through the RETURN clause. Following these guidelines will make it easier to understand and use those subprograms. In addition, functions with OUT or IN OUT parameters may not be called from within a SQL statement.

Explicit Association of Actual and Formal Parameters in PL/SQL

How does PL/SQL know which actual parameter goes with which formal parameter when a program is executed? PL/SQL offers two ways to make the association:

Positional notation
: Associates the actual parameter implicitly (by position) with the formal parameter

Named notation
: Associates the actual parameter explicitly with the formal parameter, using the formal parameter's name and the => combination symbol

Positional notation

In every example so far, I have employed positional notation to guide PL/SQL through the parameters. With positional notation, PL/SQL relies on the relative positions of the parameters to make the correspondence: it associates the *N*th actual parameter in the call to a program with the *N*th formal parameter in the program's header.

With the following total_sales example, PL/SQL associates the first actual parameter, l_company_id, with the first formal parameter, company_id_in. It then associates the second actual parameter, 'N', with the second formal parameter, status_in:

```
new_sales := total_sales (l_company_id, 'N');

FUNCTION total_sales
   (company_id_in IN company.company_id%TYPE,
    status_in IN order.status_code%TYPE := NULL)
RETURN std_types.dollar_amount;
```

Positional notation, shown in Figure 17-3, is the most common method for passing arguments to programs.

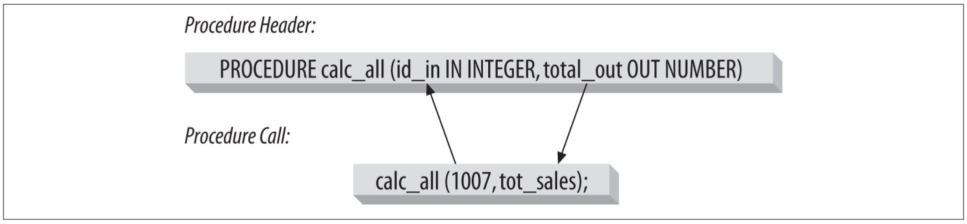

Procedure Header:
PROCEDURE calc_all (id_in IN INTEGER, total_out OUT NUMBER)
Procedure Call:
calc_all (1007, tot_sales);

Figure 17-3. Matching actual with formal parameters (positional notation)

Named notation

With named notation, you explicitly associate the formal parameter (the name of the parameter) with the actual parameter (the value of the parameter) right in the call to the program, using the combination symbol =>.

The general syntax for named notation is:

```
formal_parameter_name => argument_value
```

Because you provide the name of the formal parameter explicitly, PL/SQL no longer needs to rely on the order of the parameters to make the association from actual to formal. So, if you use named notation, you do not need to list the parameters in your call to the program in the same order as the formal parameters in the header. You can call total_sales for new orders in either of these two ways:

```
new_sales :=
   total_sales (company_id_in => order_pkg.company_id, status_in =>'N');
```

```
 new_sales :=
   total_sales (status_in =>'N', company_id_in => order_pkg.company_id);
```

You can also mix named and positional notation in the same program call:

```
l_new_sales := total_sales (order_pkg.company_id, status_in =>'N');
```

If you do mix notation, however, you must list all of your positional parameters before any named notation parameters, as shown in the preceding example. Positional notation has to have a starting point from which to keep track of positions, and the only starting point is the first parameter. If you place named notation parameters in front of positional notation, PL/SQL loses its place. Both of the following calls to total_sales will fail. The first statement fails because the named notation comes first. The second fails because positional notation is used, but the parameters are in the wrong order. In this case, PL/SQL will try to convert 'N' to a NUMBER (for company_id):

```
l_new_sales := total_sales (company_id_in => order_pkg.company_id, 'N');
l_new_sales := total_sales ('N', company_id_in => order_pkg.company_id);
```

Benefits of named notation

Now that you are aware of the different ways to notate the order and association of parameters, you might be wondering why you might ever use named notation. Here are two possibilities:

Named notation is self-documenting
: When you use named notation, the call to the program clearly describes the formal parameter to which the actual parameter is assigned. The names of formal parameters can and should be designed so that their purpose is self-explanatory. In a way, the descriptive aspect of named notation is another form of program documentation. If you are not familiar with all of the modules called by an application, the listing of the formal parameters helps reinforce your understanding of a particular program call. In some development environments, the standard for parameter notation is named notation for just this reason. This is especially true when the formal parameters are named following the convention of appending the passing mode as the last token. Then, you can clearly see the direction of data simply by investigating the procedure or function call.

Named notation gives you complete flexibility in parameter specification
: With named notation, you can list the parameters in any order you want. (This does not mean, however, that you should randomly order your arguments when you call a program!) You can also include only the parameters you want or need in the parameter list. Complex applications may at times require procedures with literally dozens of parameters. When you use named notation, any parameter with a default value can be left out of the call to the procedure; the developer can use the procedure by passing only the values needed for that usage.

Let's see how these benefits can be applied. Consider the following program header:

```
/* File on web: namednot.sql */
PROCEDURE business_as_usual (
   advertising_budget_in   IN     NUMBER
 , contributions_inout     IN OUT NUMBER
 , merge_and_purge_on_in   IN     DATE DEFAULT SYSDATE
 , obscene_ceo_bonus_out   OUT    NUMBER
 , cut_corners_in          IN     VARCHAR2 DEFAULT 'WHENEVER POSSIBLE'
);
```

An analysis of the parameter list yields these conclusions:

- The minimum number of arguments that must be passed to business_as_usual is three. To determine this, add the number of IN parameters without default values to the number of OUT and IN OUT parameters.
- You can call this program with positional notation with either four or five arguments, because the last parameter has mode IN with a default value.
- You will need at least two variables to hold the values returned by the OUT and IN OUT parameters.

Given this parameter list, there are a number of ways that you can call this program:

- All positional notation, all actual parameters specified. Notice how difficult it is to recall the formal parameters associated with (and significance of) each of these values:

  ```
  DECLARE
     l_ceo_payoff         NUMBER;
     l_lobbying_dollars   NUMBER := 100000;
  BEGIN
     /* All positional notation */
     business_as_usual (50000000
                      , l_lobbying_dollars
                      , SYSDATE + 20
                      , l_ceo_payoff
                      , 'PAY OFF OSHA'
                       );
  ```

- All positional notation, minimum number of actual parameters specified. This is still hard to understand:

  ```
  business_as_usual (50000000
                   , l_lobbying_dollars
                   , SYSDATE + 20
                   , l_ceo_payoff
                    );
  ```

- All named notation, keeping the original order intact. Now the call to business_as_usual is self-documenting:

  ```
  business_as_usual
    (advertising_budget_in      => 50000000
  ```

```
  , contributions_inout        => l_lobbying_dollars
  , merge_and_purge_on_in      => SYSDATE
  , obscene_ceo_bonus_out      => l_ceo_payoff
  , cut_corners_in             => 'DISBAND OSHA'
   );
```

- Skipping over all IN parameters with default values (another critical feature of named notation):

```
business_as_usual
   (advertising_budget_in      => 50000000
  , contributions_inout        => l_lobbying_dollars
  , obscene_ceo_bonus_out      => l_ceo_payoff
  );
```

- Changing the order in which actual parameters are specified with named notation, and providing just a partial list:

```
business_as_usual
  (obscene_ceo_bonus_out      => l_ceo_payoff
 , merge_and_purge_on_in      => SYSDATE
 , advertising_budget_in      => 50000000
 , contributions_inout        => l_lobbying_dollars
  );
```

- Blending positional and named notation. You can start with positional, but once you switch to named notation, you can't go back to positional:

```
business_as_usual
   (50000000
   , l_lobbying_dollars
   , merge_and_purge_on_in      => SYSDATE
   , obscene_ceo_bonus_out      => l_ceo_payoff
    );
```

As you can see, there is lots of flexibility when it comes to passing arguments to a parameter list in PL/SQL. As a general rule, named notation is the best way to write code that is readable and more easily maintained. You just have to take the time to look up and write the parameter names.

The NOCOPY Parameter Mode Qualifier

PL/SQL offers an option for modifying the definition of a parameter: the NOCOPY clause. NOCOPY requests that the PL/SQL compiler *not* make copies of OUT and IN OUT arguments—and under most circumstances, the compiler will grant that request. The main objective of using NOCOPY is to improve the performance of passing large constructs, such as collections, as IN OUT arguments. Because of its performance implications, this topic is covered in detail in Chapter 21.

Default Values

As you have seen from previous examples, you can provide default values for IN parameters. If an IN parameter has a default value, you do not need to include that parameter in the call to the program. Likewise, a parameter's default value is evaluated and used by the program only if the call to that program does not include that parameter in the list. You must, of course, include an actual parameter for any IN OUT parameters.

Specifying a default value for a parameter works the same way as specifying a default value for a declared variable. There are two ways to specify a default value—either with the keyword DEFAULT or with the assignment operator (:=)—as the following example illustrates:

```
PROCEDURE astrology_reading
   (sign_in IN VARCHAR2 := 'LIBRA',
    born_at_in IN DATE DEFAULT SYSDATE) IS
```

Using default values allows you to call programs with different numbers of actual parameters. The program uses the default values of any unspecified parameters, and overrides the default values of any parameters in the list that have specified values. Here are all the different ways you can ask for your astrology reading using positional notation:

```
BEGIN
   astrology_reading ('SCORPIO',
      TO_DATE ('12-24-2009 17:56:10', 'MM-DD-YYYY HH24:MI:SS'));
   astrology_reading ('SCORPIO');
   astrology_reading;
   astrology_reading();
END;
```

The first call specifies both parameters explicitly. In the second call, only the first actual parameter is included, so born_at_in is set to the current date and time. In the third call, no parameters are specified, so I omit the parentheses (the same goes for the final call, where I specify empty parentheses). Both of the default values are used in the body of the procedure.

What if you want to specify a birth time, but not a sign? To skip over leading parameters that have default values, you will need to use named notation. By including the names of the formal parameters, you can list only those parameters to which you need to pass values. In this (thankfully) last request for a star-based reading of my fate, I have successfully passed in a default of Libra as my sign and an overridden birth time of 5:56 p.m.:

```
BEGIN
   astrology_reading (
      born_at_in =>
         TO_DATE ('12-24-2009 17:56:10', 'MM-DD-YYYY HH24:MI:SS'));
END;
```

Local or Nested Modules

A *local* or *nested module* is a procedure or function that is defined in the declaration section of a PL/SQL block (anonymous or named). This module is considered local because it is defined only within the parent PL/SQL block. It cannot be called by any other PL/SQL blocks defined outside that enclosing block.

Figure 17-4 shows how blocks that are external to a procedure definition cannot “cross the line” into the procedure to directly invoke any local procedures or functions.

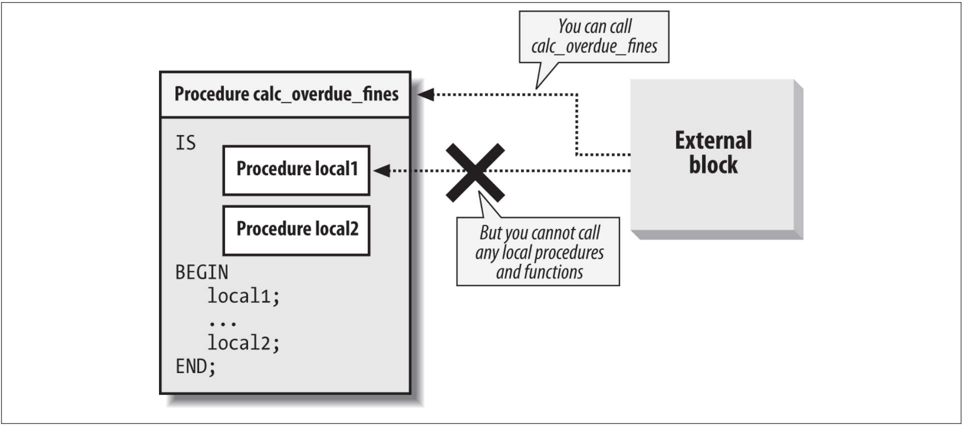

You can call calc_overdue_fines
Procedure calc_overdue_fines
IS
Procedure local1
Procedure local2
BEGIN
local1;
...
local2;
END;
External block
But you cannot call any local procedures and functions

Figure 17-4. Local modules are hidden and inaccessible outside the program

The syntax for defining the procedure or function is exactly the same as that used for creating standalone modules.

The following anonymous block, for example, declares a local procedure:

```
DECLARE
   PROCEDURE show_date (date_in IN DATE) IS
   BEGIN
      DBMS_OUTPUT.PUT_LINE (TO_CHAR (date_in, 'Month DD, YYYY');
   END show_date;
BEGIN
   ...
END ;
```

Local modules must be located after all of the other declaration statements in the declaration section. You must declare your variables, cursors, exceptions, types, records, tables, and so on before you type in the first PROCEDURE or FUNCTION keyword.

The following sections explore the benefits of local modules and offer a number of examples.

Benefits of Local Modularization

There are two central reasons to create local modules:

To reduce the size of the module by stripping it of repetitive code
: This is the most common motivation to create a local module; you can see its impact in the next example. The code reduction leads to higher code quality because you have fewer lines to test and fewer potential bugs. It takes less effort to maintain the code because there is less to maintain. And when you do have to make a change, you make it in one place in the local module, and the effects are felt immediately throughout the parent module.

To improve the readability of your code
: Even if you do not repeat sections of code within a module, you still may want to pull out a set of related statements and package them into a local module. This can make it easier to follow the logic of the main body of the parent module.

The following sections examine these benefits.

Reducing code volume

Let's look at an example of reducing code volume. The calc_percentages procedure takes numeric values from the sales package (sales_pkg), calculates the percentage of each sales amount against the total sales provided as a parameter, and then formats the number for display in a report or form. The example you see here has only 3 calculations, but I extracted it from a production application that actually performed 23 of these computations!

```
PROCEDURE calc_percentages (total_sales_in IN NUMBER)
IS
   l_profile sales_descriptors%ROWTYPE;
BEGIN
   l_profile.food_sales_stg :=
      TO_CHAR ((sales_pkg.food_sales / total_sales_in ) * 100,
               '$999,999');
   l_profile.service_sales_stg :=
      TO_CHAR ((sales_pkg.service_sales / total_sales_in ) * 100,
               '$999,999');
   l_profile.toy_sales_stg :=
      TO_CHAR ((sales_pkg.toy_sales / total_sales_in ) * 100,
               '$999,999');
END;
```

This code took a long time (relatively speaking) to write, is larger than necessary, and is maintenance-intensive. What if I need to change the format to which I convert the numbers? What if the calculation of the percentage changes? I will have to change each of the individual calculations.

With local modules, I can concentrate all the common, repeated code into a single function, which is then called repeatedly in calc_percentages. The local module version of this procedure is shown here:

```
PROCEDURE calc_percentages (total_sales_in IN NUMBER)
IS
   l_profile sales_descriptors%ROWTYPE;
   /* Define a function right inside the procedure! */
   FUNCTION pct_stg (val_in IN NUMBER) RETURN VARCHAR2
   IS
   BEGIN
      RETURN TO_CHAR ((val_in/total_sales_in ) * 100, '$999,999');
   END;
BEGIN
   l_profile.food_sales_stg := pct_stg (sales_pkg.food_sales);
   l_profile.service_sales_stg := pct_stg (sales_pkg.service_sales);
   l_profile.toy_sales_stg := pct_stg (sales_pkg.toy_sales);
END;
```

All of the complexities of the calculation, from the division by total_sales_in to the multiplication by 100 to the formatting with TO_CHAR, have been transferred to the function pct_stg. This function is defined in the declaration section of the procedure. Calling this function from within the body of calc_percentages makes the executable statements of the procedure much more readable and maintainable. Now, if the formula for the calculation changes in any way, I can make the change just once in the function and it will take effect in all the assignments.

Improving readability

You can use local modules to dramatically improve the readability and maintainability of your code. In essence, local modules allow you to follow *top-down design* or *stepwise refinement* methodologies very closely. You can also use the same technique to *decompose* or *refactor* an existing program so that it is more readable.

The bottom-line result of using local modules in this way is that you can dramatically reduce the size of your execution sections (you are transferring many lines of logic from an inline location in the execution section to a local module callable in that section). If you keep your execution sections small, you will find that it is much easier to read and understand the logic.

I suggest that you adopt as a guideline in your coding standards that execution sections of PL/SQL blocks be no longer than 60 lines (the amount of text that can fit on a screen or page). This may sound crazy, but if you follow the techniques in this section, you will find it not only possible but highly advantageous.

Suppose that I have a series of WHILE loops (some of them nested) whose bodies contain a series of complex calculations and deep nestings of conditional logic. Even with extensive commenting, it can be difficult to follow the program flow over several pages, particularly when the END IF or END LOOP of a given construct is not even on the same page as the IF or LOOP statement that began it.

In contrast, if I pull out sequences of related statements, place them in one or more local modules, and then call those modules in the body of the program, the result is a program that can literally document itself. The assign_workload procedure offers a simplified version of this scenario that still makes clear the gains offered by local modules:

```
/* File on web: local_modules.sql */
PROCEDURE assign_workload (department_in IN emp.deptno%TYPE)
IS
   CURSOR emps_in_dept_cur (department_in IN emp.deptno%TYPE)
   IS
      SELECT * FROM emp WHERE deptno = department_in;

   PROCEDURE assign_next_open_case
      (emp_id_in IN NUMBER, case_out OUT NUMBER)
   IS
   BEGIN ...  full implementation ... END;

   FUNCTION next_appointment (case_id_in IN NUMBER)
      RETURN DATE
   IS
   BEGIN ...  full implementation ... END;

   PROCEDURE schedule_case
       (case_in IN NUMBER, date_in IN DATE)
   IS
   BEGIN ...  full implementation ... END;

BEGIN /* main */
   FOR emp_rec IN emps_in_dept_cur (department_in)
   LOOP
      IF analysis.caseload (emp_rec.emp_id) <
         analysis.avg_cases (department_in);
      THEN
         assign_next_open_case (emp_rec.emp_id, case#);
         schedule_case
            (case#, next_appointment (case#));
      END IF;
   END LOOP
END assign_workload;
```

The assign_workload procedure has three local modules:

```
assign_next_open_case
next_appointment
schedule_case
```

It also relies on two packaged programs that already exist and can be easily plugged into this program: analysis.caseload and analysis.avg_cases. For the purposes of understanding the logic behind assign_workload, it doesn't really matter what code is executed in each of them. I can rely simply on the names of those modules to read through the main body of this program. Even without any comments, a reader can still gain a clear understanding of what each module is doing. Of course, if you want to rely on named objects to self-document your code, you'd better come up with very good names for the functions and procedures!

Scope of Local Modules

The modularized declaration section looks a lot like the body of a package, as you will see in Chapter 18. A package body also contains definitions of modules. The big difference between local modules and package modules is their scope. Local modules can be called only from within the block in which they are defined; package modules can—at a minimum—be called from anywhere in the package. If the package modules are also listed in the package specification, they can be called by other program units from schemas that have EXECUTE authority on that package.

You should therefore use local modules only to encapsulate code that does not need to be called outside of the current program. Otherwise, go ahead and create a package!

Sprucing Up Your Code with Nested Subprograms

These days it seems that whenever I write a program with more than 20 lines and any complexity whatsoever, I end up creating one or more local modules. Doing so helps me see my way through to a solution much more easily; I can conceptualize my code at a higher level of abstraction by assigning a name to a whole sequence of statements, and I can perform top-down design and stepwise refinement of my requirements. Finally, by modularizing my code even within a single program, I make it very easy to later extract a nested subprogram and make it a truly independent, reusable procedure or function.

You could also, of course, move that logic out of the local scope and make it a package body–level program of its own (assuming you are writing this code in a package). Taking this approach will reduce the amount of nesting of local procedures, which can be helpful. It also, however, can lead to package bodies with very long lists of programs, many of which are only used within another program. My general principle is to keep the definition of an element as close as possible to its usage, which naturally leads to the use of nested subprograms.

I hope that as you read this, a program you have written comes to mind. Perhaps you can go back and consolidate some repetitive code, clean up the logic, and make the program actually understandable to another human being. Don't fight the urge. Go ahead and modularize your code.

To help you define and work with nested subprograms in your applications, I have created a package called TopDown. Using this package, you can spend a small amount of time placing "indicators" in your code, which are essentially instructions on what nested subprograms you want created, and how. You can then compile this sort-of-template into the database, call TopDown.Refactor for that program unit, and voilà!—nested subprograms are created as you requested.

You can then repeat that process for each level down through the complexities of your program, very quickly defining a highly modular architecture that you and others will appreciate for years to come.

You will find a more complete explanation of the TopDown package, the source code, and example scripts in the *TopDown.zip* file on the book's website.

Subprogram Overloading

When more than one subprogram in the same scope shares the same name, the subprograms are said to be *overloaded*. PL/SQL supports the overloading of procedures and functions in the declaration section of a block (named or anonymous), package specifications and bodies, and object type definitions. Overloading is a very powerful feature, and you should exploit it fully to improve the usability of your software.

Here is a very simple example of three overloaded subprograms defined in the declaration section of an anonymous block (therefore, all are local modules):

```
DECLARE
   /* First version takes a DATE parameter. */
   FUNCTION value_ok (date_in IN DATE) RETURN BOOLEAN IS
   BEGIN
      RETURN date_in <= SYSDATE;
   END;

   /* Second version takes a NUMBER parameter. */
   FUNCTION value_ok (number_in IN NUMBER) RETURN BOOLEAN  IS
   BEGIN
      RETURN number_in > 0;
   END;

   /* Third version is a procedure! */
   PROCEDURE value_ok (number_in IN NUMBER) IS
   BEGIN
      IF number_in > 0 THEN
         DBMS_OUTPUT.PUT_LINE (number_in || 'is OK!');
      ELSE
         DBMS_OUTPUT.PUT_LINE (number_in || 'is not OK!');
      END IF;
   END;

BEGIN
```

When the PL/SQL runtime engine encounters the following statement:

```
IF value_ok (SYSDATE) THEN ...
```

the actual parameter list is compared with the formal parameter lists of the various overloaded modules, searching for a match. If one is found, PL/SQL executes the code in the body of the program with the matching header.

> Another name for overloading is *static polymorphism*. The term *polymorphism* refers to the ability of a language to define and selectively use more than one form of a program with the same name. When the decision on which form to use is made at compilation time, it is called static polymorphism. When the decision is made at runtime, it is called *dynamic polymorphism*; this type of polymorphism is available through inherited object types.

Overloading can greatly simplify your life and the lives of other developers. This technique consolidates the call interfaces for many similar programs into a single module name, transferring the burden of knowledge from the developer to the software. You do not have to try to remember, for instance, the six different names for programs adding values (dates, strings, Booleans, numbers, etc.) to various collections. Instead, you simply tell the compiler that you want to add a value and pass it that value. PL/SQL and your overloaded programs figure out what you want to do and then do it for you.

When you build overloaded subprograms, you spend more time in design and implementation than you might with separate, standalone programs. This additional time up front will be repaid handsomely down the line because you and others will find it much easier and more efficient to use your programs.

Benefits of Overloading

There are three different scenarios that benefit from overloading:

Supporting many data combinations
: When you are applying the same action to different kinds or combinations of data, overloading does not provide a single name for different activities so much as it provides different ways of requesting the same activity. This is the most common motivation for overloading.

Fitting the program to the user
: To make your code as useful as possible, you may construct different versions of the same program that correspond to different patterns of use. This often involves overloading functions and procedures. A good indicator of the need for this form of overloading is when you find yourself writing unnecessary code. For example, when working with DBMS_SQL, you will call the DBMS_SQL.EXECUTE function, but for DDL statements, the value returned by this function is irrelevant. Oracle

should have overloaded this function as a procedure, so that you could simply execute a DDL statement like this:

```
BEGIN
   DBMS_SQL.EXECUTE ('CREATE TABLE xyz ...');
```

as opposed to writing code like this:

```
DECLARE
   feedback PLS_INTEGER;
BEGIN
   feedback := DBMS_SQL.EXECUTE ('CREATE TABLE xyz ...');
```

and then ignoring the feedback.

Overloading by type, not value

This is the least common application of overloading. In this scenario, you use the type of data, not its value, to determine which of the overloaded programs should be executed. This really comes in handy only when you are writing very generic software. DBMS_SQL.DEFINE_COLUMN is a good example of this approach to overloading. You need to tell DBMS_SQL the type of each of your columns being selected from the dynamic query. To indicate a numeric column, you can make a call as follows:

```
DBMS_SQL.DEFINE_COLUMN (cur, 1, 1);
```

or you could do this:

```
DBMS_SQL.DEFINE_COLUMN (cur, 1, DBMS_UTILITY.GET_TIME);
```

It doesn't matter which you do; you just need to say, "this is a number," but not any particular number. Overloading is an elegant way to handle this requirement.

Let's look at an example of the most common type of overloading and then review restrictions and guidelines on overloading.

Supporting many data combinations

You can use overloading to apply the same action to different kinds or combinations of data. As noted previously, this kind of overloading does not provide a single name for different activities so much as different ways of requesting the same activity. Consider DBMS_OUTPUT.PUT_LINE. You can use this built-in to display the value of any type of data that can be implicitly or explicitly converted to a string. Interestingly, in earlier versions of Oracle Database (7, 8, 8*i*, 9*i*), this procedure was overloaded. In Oracle Database 10*g* and later, however, it is not overloaded at all! This means that if you want to display an expression that cannot be implicitly converted to a string, you cannot call DBMS_OUTPUT.PUT_LINE and pass it that expression.

You might be thinking: so what? PL/SQL implicitly converts numbers and dates to a string. What else might I want to display? Well, for starters, how about a Boolean? To display the value of a Boolean expression, you must write an IF statement, as in:

```
IF l_student_is_registered
THEN
   DBMS_OUTPUT.PUT_LINE ('TRUE');
ELSE
   DBMS_OUTPUT.PUT_LINE ('FALSE');
END IF;
```

Now, isn't that silly? And a big waste of your time? Fortunately, it is very easy to fix this problem. Just build your own package, with lots of overloadings, on top of DBMS_OUTPUT.PUT_LINE. Here is a very abbreviated example of such a package. You can extend it easily, as I do with the do.pl procedure (why type all those characters just to say "show me," right?). A portion of the package specification is shown here:

```
/* File on web: do.pkg (also check out the p.* files) */
PACKAGE do
IS
   PROCEDURE pl (boolean_in IN BOOLEAN);

   /* Display a string. */
   PROCEDURE pl (char_in IN VARCHAR2);

   /* Display a string and then a Boolean value. */
   PROCEDURE pl (
      char_in      IN   VARCHAR2,
      boolean_in   IN   BOOLEAN
   );

   PROCEDURE pl (xml_in IN SYS.XMLType);
END do;
```

This package simply sits on top of DBMS_OUTPUT.PUT_LINE and enhances it. With do.pl, I can now display a Boolean value without writing my own IF statement, as in:

```
DECLARE
   v_is_valid BOOLEAN :=
      book_info.is_valid_isbn ('5-88888-66');
BEGIN
   do.pl (v_is_valid);
```

Better yet, I can get really fancy and even apply do.pl to complex datatypes like XMLType:

```
/* File on web: xmltype.sql */
DECLARE
   doc   xmltype;
BEGIN
   SELECT ea.report
   INTO doc
   FROM env_analysis ea
  WHERE company= 'ACME SILVERPLATING';
```

```
 do.pl (doc);
END;
```

Restrictions on Overloading

There are several restrictions on how you can overload programs. When the PL/SQL engine compiles and runs your program, it has to be able to distinguish between the different overloaded versions; after all, it can't run two different modules at the same time. So when you compile your code, PL/SQL will reject any improperly overloaded modules. It cannot distinguish between the modules by their names, because by definition they are the same in all overloaded versions of a program. Instead, PL/SQL uses the parameter lists of these sibling programs to determine which one to execute and/or the types of the programs (procedure versus function). As a result, the following restrictions apply to overloaded programs:

The datatype "family" of at least one of the parameters of the overloaded programs must differ

INTEGER, REAL, DECIMAL, FLOAT, and so on are NUMBER subtypes. CHAR, VARCHAR2, and LONG are character subtypes. If the parameters differ by datatype but only within a given supertype or family of datatypes, PL/SQL does not have enough information to determine the appropriate program to execute.

However, see the following section, which explains an improvement in Oracle Database 10*g* (and later) regarding overloading for numeric types.

Overloaded programs with parameter lists that differ only by name must be called using named notation

If you don't use the name of the argument, how can the compiler distinguish between calls to two overloaded programs? Please note, however, that it is always risky to use named notation as an enforcement paradigm. You should avoid situations where named notation yields different semantic meaning from positional notation.

The parameter lists of overloaded programs must differ by more than parameter mode

Even if a parameter in one version is IN and that same parameter in another version is IN OUT, PL/SQL cannot tell the difference at the point at which the program is called.

All of the overloaded programs must be defined within the same PL/SQL scope or block (anonymous block, standalone procedure or function, or package)

You cannot define one version in one block (scope level) and define another version in a different block. You cannot overload two standalone programs; one simply replaces the other.

Overloaded functions must differ by more than their return type (the datatype specified in the RETURN clause of the function)

At the time that the overloaded function is called, the compiler doesn't know what type of data that function will return. The compiler therefore cannot determine which version of the function to use if all the parameters are the same.

Overloading with Numeric Types

Starting with Oracle Database 10*g*, you can overload two subprograms if their formal parameters differ only in numeric datatype. Before getting into the details, let's look at an example. Consider the following block:

```
DECLARE
   PROCEDURE proc1 (n IN PLS_INTEGER) IS
   BEGIN
      DBMS_OUTPUT.PUT_LINE ('pls_integer version');
   END;

   PROCEDURE proc1 (n IN NUMBER) IS
   BEGIN
      DBMS_OUTPUT.PUT_LINE ('number version');
   END;
BEGIN
   proc1 (1.1);
   proc1 (1);
END;
```

When I try to run this code in Oracle9*i* Database, I get an error:

```
ORA-06550: line 14, column 4:
PLS-00307: too many declarations of 'PROC1' match this call
```

When I run this same block in Oracle Database 10*g* and Oracle Database 11*g*, however, I see the following results:

```
number version
pls_integer version
```

The PL/SQL compiler is now able to distinguish between the two calls. Notice that it called the "number version" when I passed a noninteger value. That's because PL/SQL looks for numeric parameters that match the value, and it follows this order of precedence in establishing the match: it starts with PLS_INTEGER or BINARY_INTEGER, then NUMBER, then BINARY_FLOAT, and finally BINARY_DOUBLE. It will use the first overloaded program that matches the actual argument values passed.

While it is very nice that the database now offers this flexibility, be careful when relying on this very subtle overloading—make sure that it is all working as you would expect. Test your code with a variety of inputs and check the results. Remember that you can pass a string such as "156.4" to a numeric parameter; be sure to try out those inputs as well.

You can also qualify numeric literals and use conversion functions to make explicit which overloading (i.e., which numeric datatype) you want to call. If you want to pass 5.0 as a BINARY_FLOAT, for example, you could specify the value 5.0f or use the conversion function TO_BINARY_FLOAT(5.0).

Forward Declarations

PL/SQL requires that you declare elements before using them in your code. Otherwise, how can PL/SQL be sure that the way you are using the construct is appropriate? Because modules can call other modules, however, you may encounter situations where it is completely impossible to define all modules before any references to those modules are made. What if program A calls program B and program B calls program A? PL/SQL supports *recursion*, including mutual recursion, in which two or more programs directly or indirectly call each other.

If you find yourself committed to mutual recursion, you will be very glad to hear that PL/SQL supports the *forward declaration* of local modules, which means that modules are declared in advance of the actual definition of the programs. This declaration makes a program available to be called by other programs even before the program is defined.

Remember that both procedures and functions have a header and a body. A forward declaration consists simply of the program header followed by a semicolon (;). This construction is called the *module header*. This header, which must include the parameter list (and a RETURN clause if it's a function), is all the information PL/SQL needs about a module in order to declare it and resolve any references to it.

The following example illustrates the technique of forward declaration. I define two mutually recursive functions within a procedure. Consequently, I have to declare just the header of my second function, total_cost, before the full declaration of net_profit:

```
PROCEDURE perform_calcs (year_in IN INTEGER)
IS
   /* Header only for total_cost function. */
   FUNCTION total_cost (...)  RETURN NUMBER;

   /* The net_profit function uses total_cost. */
   FUNCTION net_profit (...) RETURN NUMBER    IS
   BEGIN
      RETURN total_sales (...) - total_cost (...);
   END;
   /* The total_cost function uses net_profit. */
   FUNCTION total_cost (...)  RETURN NUMBER    IS
   BEGIN
      IF <condition based on parameters>
      THEN
         RETURN net_profit (...) * .10;
      ELSE
         RETURN <parameter value>;
      END IF;
   END;
BEGIN
   ...
END;
```

Here are some rules to remember concerning forward declarations:

- You cannot make forward declarations of a variable or cursor. This technique works only with modules (procedures and functions).
- The definition for a forwardly declared program must be contained in the declaration section of the same PL/SQL block (anonymous block, procedure, function, or package body) in which you code the forward declaration.

In some situations, forward declarations are absolutely required; in most situations, they just help make your code more readable and presentable. As with every other advanced or unusual feature of the PL/SQL language, use forward declarations only when you really need the functionality. Otherwise, the declarations simply add to the clutter in your program, which is the last thing you want.

Advanced Topics

The following sections are most appropriate for experienced PL/SQL programmers. Here, I'll touch on a number of advanced modularization topics, including calling functions in SQL, using table functions, and using deterministic functions.

Calling Your Function from Inside SQL

The Oracle database allows you to call your own custom-built functions from within SQL. In essence, this flexibility allows you to customize the SQL language to adapt to application-specific requirements.

Whenever the SQL runtime engine calls a PL/SQL function, it must "switch" to the PL/SQL runtime engine. The overhead of this context switch can be substantial if the function is called many times.

Requirements for calling functions in SQL

There are several requirements that a programmer-defined PL/SQL function must meet in order to be callable from within a SQL statement:

- All of the function's parameters must use the IN mode. Neither IN OUT nor OUT parameters are allowed in SQL-embedded stored functions.
- The datatypes of the function's parameters, as well as the datatype of the RETURN clause of the function, must be SQL datatypes or types that can be implicitly converted to them. While all of the Oracle server datatypes are valid within PL/SQL, PL/SQL has added new datatypes that are not (yet) supported in the database. These datatypes include BOOLEAN, BINARY_INTEGER, associative arrays, PL/SQL records, and programmer-defined subtypes.
- The function must be stored in the database. A function defined in a client-side PL/SQL environment cannot be called from within SQL; there would be no way for SQL to resolve the reference to the function.

By default, user-defined functions that execute in SQL operate on a single row of data, not on an entire column of data that crosses rows, as the group functions SUM, MIN, and AVG do. It is possible to write aggregate functions to be called inside SQL, but this requires taking advantage of the ODCIAggregate interface, which is part of Oracle's Extensibility Framework. See the Oracle documentation for more details on this functionality.

Restrictions on user-defined functions in SQL

In order to guard against nasty side effects and unpredictable behavior, the Oracle database applies many restrictions on what you can do from within a user-defined function executed inside a SQL statement:

- The function may not modify database tables. It may not execute any of the following types of statements: DDL (CREATE TABLE, DROP INDEX, etc.), INSERT, DELETE, MERGE, or UPDATE. Note that this restriction is relaxed if your function is defined as an autonomous transaction (described in Chapter 14); in this case, any changes made in your function occur independently of the outer transaction in which the query was executed.
- When called remotely or through a parallelized action, the function may not read or write the values of package variables. The Oracle server does not support side effects that cross user sessions.

- The function can update the values of package variables only if that function is called in a select list, or a VALUES or SET clause. If the stored function is called in a WHERE or GROUP BY clause, it may not write package variables.
- Prior to Oracle8 Database, you may not call RAISE_APPLICATION_ERROR from within the user-defined function.
- The function may not call another module (stored procedure or function) that breaks any of the preceding rules. A function is only as pure as the most impure module that it calls.
- The function may not reference a view that breaks any of the preceding rules. A view is a stored SELECT statement; that view's SELECT may use stored functions.
- Prior to Oracle Database 11*g* and later, you may use only positional notation to pass actual arguments to your function's formal parameters. In Oracle Database 11*g*, you may use named and mixed notation.

Read consistency and user-defined functions

The read consistency model of the Oracle database is simple and clear: once I start a query, that query will only see data as it existed (was committed in the database) at the time the query was started, as well as the results of any changes made by DML statements in my current transaction. So, if my query starts at 9:00 a.m. and runs for an hour, then even if another user comes along and changes data during that period, my query will not see those changes.

Yet unless you take special precautions with user-defined functions in your queries, it is quite possible that your query will violate (or, at least, appear to violate) the read consistency model of the Oracle database. To understand this issue, consider the following function and the query that calls it:

```
FUNCTION total_sales (id_in IN account.account_id%TYPE)
   RETURN NUMBER
IS
   CURSOR tot_cur
   IS
      SELECT SUM (sales) total
        FROM orders
       WHERE account_id = id_in
         AND TO_CHAR (ordered_on, 'YYYY') = TO_CHAR (SYSDATE, 'YYYY');
   tot_rec tot_cur%ROWTYPE;
BEGIN
   OPEN tot_cur;
   FETCH tot_cur INTO tot_rec;
   CLOSE tot_cur;
   RETURN tot_rec.total;
END;
```

```
SELECT name, total_sales (account_id)
  FROM account
 WHERE status = 'ACTIVE';
```

The account table has 5 million active rows in it (a very successful enterprise!). The orders table has 20 million rows. I start the query at 10:00 a.m.; it takes about an hour to complete. At 10:45 a.m., somebody with the proper authority comes along, deletes all rows from the orders table, and performs a commit. According to the read consistency model of Oracle, the session running the query should not see that all those rows have been deleted until the query completes. But the next time the total_sales function executes from within the query, it will find no order rows and return NULL—and will do so until the query completes.

So, if you are executing queries inside functions that are called inside SQL, you need to be acutely aware of read consistency issues. If these functions are called in long-running queries or transactions, you will probably need to issue the following command to enforce read consistency *between* SQL statements in the current transaction:

```
SET TRANSACTION READ ONLY
```

In this case, for read consistency to be possible, you need to ensure that you have sufficient undo tablespace.

Defining PL/SQL subprograms in SQL statements (12.1 and higher)

Developers have long been able to call their own PL/SQL functions from within a SQL statement. Suppose, for example, I have created a function named BETWNSTR that returns the substring between the specified start and end locations:

```
FUNCTION betwnstr (
   string_in      IN   VARCHAR2
 , start_in       IN   PLS_INTEGER
 , end_in         IN   PLS_INTEGER
)
   RETURN VARCHAR2
IS
BEGIN
   RETURN ( SUBSTR (
        string_in, start_in, end_in - start_in + 1 ));
END;
```

I can then use it in a query as follows:

```
SELECT betwnstr (last_name, 3, 5)
  FROM employees
```

This feature offers a way to both "extend" the SQL language with application-specific functionality and reuse (rather than copy) algorithms. A downside of user-defined function execution in SQL is that it involves a context switch between the SQL and PL/SQL execution engines.

With Oracle Database 12*c*, you can now define PL/SQL functions and procedures in the WITH clause of a subquery, and then use it as you would any built-in or user-defined function. This feature allows me to consolidate the function and query just shown into a single statement:

```
WITH
 FUNCTION betwnstr (
     string_in   IN VARCHAR2,
     start_in    IN PLS_INTEGER,
     end_in      IN PLS_INTEGER)
 RETURN VARCHAR2
 IS
 BEGIN
   RETURN (SUBSTR (
       string_in,
       start_in,
       end_in - start_in + 1));
 END;
SELECT betwnstr (last_name, 3, 5)
  FROM employees
```

The main advantage of this approach is improved performance, since the cost of a context switch from the SQL to the PL/SQL engine is greatly reduced when the function is defined in this way. Notice, however, that I also sacrifice reusability if this same logic is used in other places in my application.

There are, however, other motivations for defining functions in the WITH clause.

While you can call a packaged function in SQL, you cannot reference a constant declared in a package (unless that SQL statement is executed inside a PL/SQL block), as shown here:

```
SQL> CREATE OR REPLACE PACKAGE pkg
  2  IS
  3     year_number   CONSTANT INTEGER := 2013;
  4  END;
  5  /

Package created.

SQL> SELECT pkg.year_number FROM employees
  2   WHERE employee_id = 138
  3  /
SELECT pkg.year_number FROM employees
       *
ERROR at line 1:
ORA-06553: PLS-221: 'YEAR_NUMBER' is not a procedure or is undefined
```

The classic workaround to this problem is to define a function in the package and then call the function:

```
SQL> CREATE OR REPLACE PACKAGE pkg
     IS
        FUNCTION year_number
           RETURN INTEGER;
     END;
     /

Package created.

SQL> CREATE OR REPLACE PACKAGE BODY pkg
     IS
        c_year_number   CONSTANT INTEGER := 2013;

        FUNCTION year_number
           RETURN INTEGER
        IS
        BEGIN
           RETURN c_year_number;
        END;
     END;
     /

Package body created.

SQL> SELECT pkg.year_number
       FROM employees
      WHERE employee_id = 138
     /

YEAR_NUMBER
-----------
       2013
```

That's a lot of code and effort simply to be able to reference the constant's value in a SQL statement. And, with 12.1, it is no longer necessary. I can, instead, simply create a function in the WITH clause:

```
WITH
 FUNCTION year_number
 RETURN INTEGER
 IS
 BEGIN
   RETURN pkg.year_number;
 END;
SELECT year_number
  FROM employees
 WHERE employee_id = 138
```

You will also find in-SQL PL/SQL functions to be handy in standby read-only databases. While you will not be able to create "helper" PL/SQL functions in such a database, you *will* be able to define these functions directly inside your queries.

The WITH FUNCTION feature is a very useful enhancement to the SQL language. You should, however, ask yourself this question each time you contemplate using it: "Do I need this same functionality in multiple places in my application?"

If the answer is "yes," then you should decide if the performance improvement of using WITH FUNCTION outweighs the potential downside of needing to copy and paste this logic into multiple SQL statements.

Note that as of 12.1 you cannot execute a static SELECT statement that contains a WITH FUNCTION clause *inside a PL/SQL block*. I know that seems very strange, and I am sure it will be possible in 12.2, but for now, the following code will raise an error as shown:

```
SQL> BEGIN
  2     WITH FUNCTION full_name (fname_in IN VARCHAR2, lname_in IN VARCHAR2)
  3             RETURN VARCHAR2
  4          IS
  5          BEGIN
  6             RETURN fname_in || ' ' || lname_in;
  7          END;
  8
  9     SELECT LENGTH (full_name (first_name, last_name))
 10       INTO c
 11       FROM employees;
 12
 13     DBMS_OUTPUT.put_line ('count = ' || c);
 14  END;
 15  /
WITH FUNCTION full_name (fname_in IN VARCHAR2, lname_in IN VARCHAR2)
                 *
ERROR at line 2:
ORA-06550: line 2, column 18:
PL/SQL: ORA-00905: missing keyword
```

In addition to the WITH FUNCTION clause, 12.1 also offers the UDF pragma, which can improve the performance of PL/SQL functions executed from SQL. See Chapter 21 for details.

Table Functions

A *table function* is a function that can be called from within the FROM clause of a query, as if it were a relational table. Table functions return collections, which can then be transformed with the TABLE operator into a structure that can be queried using the SQL language. Table functions come in very handy when you need to:

- Perform very complex transformations of data, requiring the use of PL/SQL, but you need to access that data from within a SQL statement.
- Pass complex result sets back to the host (that is, non-PL/SQL) environment. You can open a cursor variable for a query based on a table function, and let the host environment fetch through the cursor variable.

Table functions open up all sorts of possibilities for PL/SQL developers, and to demonstrate some of those possibilities we will explore both streaming table functions and pipelined table functions in more detail in this chapter:

Streaming table functions
: *Data streaming* enables you to pass from one process or stage to another without having to rely on intermediate structures. Table functions, in conjunction with the CURSOR expression, enable you to stream data through multiple transformations, all within a single SQL statement.

Pipelined table functions
: These functions return a result set in *pipelined* fashion, meaning that data is returned while the function is still executing. Add the PARALLEL_ENABLE clause to a pipelined function's header, and you have a function that will execute in parallel within a parallel query.

Prior to Oracle Database 12*c*, table functions could return only nested tables and VARRAYs. From 12.1, you can also define table functions that return an integer-indexed associative array whose type is defined in a package specification.

Let's explore how to define table functions and put them to use in an application.

Calling a function in a FROM clause

To call a function from within a FROM clause, you must do the following:

- Define the RETURN datatype of the function to be a collection (either a nested table or a VARRAY).
- Make sure that all of the other parameters to the function are of mode IN and have SQL datatypes. (You cannot, for example, call a function with a Boolean or record type argument inside a query.)
- Embed the call to the function inside the TABLE operator (if you are running Oracle8*i* Database, you will also need to use the CAST operator).

Here is a simple example of a table function. First, I will create a nested table type based on an object type of pets:

```
/* File on web: pet_family.sql */
CREATE TYPE pet_t IS OBJECT (
   name    VARCHAR2 (60),
   breed   VARCHAR2 (100),
   dob     DATE);

CREATE TYPE pet_nt IS TABLE OF pet_t;
```

Now I will create a function named pet_family. It accepts two pet objects as arguments: the mother and the father. Then, based on the breed, it returns a nested table with the entire family defined in the collection:

```
FUNCTION pet_family (dad_in IN pet_t, mom_in IN pet_t)
   RETURN pet_nt
IS
   l_count PLS_INTEGER;
   retval   pet_nt := pet_nt ();

   PROCEDURE extend_assign (pet_in IN pet_t) IS
   BEGIN
      retval.EXTEND;
      retval (retval.LAST) := pet_in;
   END;
BEGIN
   extend_assign (dad_in);
   extend_assign (mom_in);

   IF    mom_in.breed = 'RABBIT'   THEN l_count := 12;
   ELSIF mom_in.breed = 'DOG'      THEN l_count := 4;
   ELSIF mom_in.breed = 'KANGAROO' THEN l_count := 1;
   END IF;

   FOR indx IN 1 .. l_count
   LOOP
      extend_assign (pet_t ('BABY' || indx, mom_in.breed, SYSDATE));
   END LOOP;

   RETURN retval;
END;
```

The pet_family function is silly and trivial; the point to understand here is that your PL/SQL function may contain extremely complex logic—whatever is required within your application and can be accomplished with PL/SQL—that exceeds the expressive capabilities of SQL.

Now I can call this function in the FROM clause of a query, as follows:

```
SELECT pets.NAME, pets.dob
 FROM TABLE (pet_family (pet_t ('Hoppy', 'RABBIT', SYSDATE)
                       , pet_t ('Hippy', 'RABBIT', SYSDATE)
```

```
                             )
                ) pets;
```

And here is a portion of the output:

```
NAME       DOB
---------- ---------
Hoppy      27-FEB-02
Hippy      27-FEB-02
BABY1      27-FEB-02
BABY2      27-FEB-02
...
BABY11     27-FEB-02
BABY12     27-FEB-02
```

Passing table function results with a cursor variable

Table functions help overcome a problem that developers have encountered in the past —namely, how do I pass data that I have produced through PL/SQL-based programming (i.e., data that is not intact inside one or more tables in the database) back to a non-PL/SQL host environment? Cursor variables allow me to easily pass back SQL-based result sets to, say, Java programs, because cursor variables are supported in JDBC. Yet if I first need to perform complex transformations in PL/SQL, how then do I offer that data to the calling program?

Now, we can combine the power and flexibility of table functions with the wide support for cursor variables in non-PL/SQL environments (explained in detail in Chapter 15) to solve this problem.

Suppose, for example, that I need to generate a pet family (bred through a call to the pet_family function, as shown in the previous section) and pass those rows of data to a frontend application written in Java. I can do this very easily as follows:

```
/* File on web: pet_family.sql */
FUNCTION pet_family_cv
   RETURN SYS_REFCURSOR
IS
   retval SYS_REFCURSOR;
BEGIN
   OPEN retval FOR
      SELECT *
        FROM TABLE (pet_family (pet_t ('Hoppy', 'RABBIT', SYSDATE)
                              , pet_t ('Hippy', 'RABBIT', SYSDATE)
                               )
                   );

   RETURN retval;
END pet_family_cv;
```

In this program, I am taking advantage of the predefined weak REF CURSOR type, SYS_REFCURSOR (introduced in Oracle9*i* Database), to declare a cursor variable. I

OPEN FOR this cursor variable, associating with it the query that is built around the pet_family table function.

I can then pass this cursor variable back to the Java frontend. Because JDBC recognizes cursor variables, the Java code can then easily fetch the rows of data and integrate them into the application.

Creating a streaming function

A *streaming function* accepts as a parameter a result set (via a CURSOR expression) and returns a result set in the form of a collection. Because you can apply the TABLE operator to this collection and then query from it in a SELECT statement, these functions can perform one or more transformations of data within a single SQL statement.

Streaming functions, support for which was added in Oracle9*i* Database, can be used to hide algorithmic complexity behind a function interface and thus simplify the SQL in your application. I will walk through an example to explain the kinds of steps you will need to go through yourself to take advantage of table functions in this way.

Consider the following scenario. I have a table of stock ticker information that contains a single row for the open and close prices of a stock:

```
/* File on web: tabfunc_streaming.sql */
TABLE stocktable (
  ticker VARCHAR2(10),
  trade_date DATE,
  open_price NUMBER,
  close_price NUMBER)
```

I need to transform (or *pivot*) that information into another table:

```
TABLE tickertable (
  ticker VARCHAR2(10),
  pricedate DATE,
  pricetype VARCHAR2(1),
  price NUMBER)
```

In other words, a single row in stocktable becomes two rows in tickertable. There are many ways to achieve this goal. A very traditional and straightforward approach in PL/SQL might look like this:

```
FOR rec IN  (SELECT * FROM stocktable)
LOOP
   INSERT INTO tickertable
               (ticker, pricetype, price)
        VALUES (rec.ticker, 'O', rec.open_price);

   INSERT INTO tickertable
               (ticker, pricetype, price)
        VALUES (rec.ticker, 'C', rec.close_price);
END LOOP;
```

There are also 100% SQL solutions, such as:

```
INSERT ALL
      INTO tickertable
           (ticker, pricedate, pricetype, price
           )
    VALUES (ticker, trade_date, 'O', open_price
           )
      INTO tickertable
           (ticker, pricedate, pricetype, price
           )
    VALUES (ticker, trade_date, 'C', close_price
           )
   SELECT ticker, trade_date, open_price, close_price
     FROM stocktable;
```

Let's assume, however, that the transformation that I must perform to move data from stocktable to tickertable is very complex and requires the use of PL/SQL. In this situation, a table function used to stream the data as it is transformed would offer a much more efficient solution.

First of all, if I am going to use a table function, I will need to return a nested table or VARRAY of data. I will use a nested table because VARRAYs require the specification of a maximum size, and I don't want to have that restriction in my implementation. This nested table type must be defined as a schema-level type or within the specification of a package, so that the SQL engine can resolve a reference to a collection of this type.

I would like to return a nested table based on the table definition itself—that is, I would like it to be defined as follows:

```
TYPE tickertype_nt IS TABLE of tickertable%ROWTYPE;
```

Unfortunately, this statement will fail because %ROWTYPE is not a SQL-recognized type. That attribute is available only inside a PL/SQL declaration section. So, I must instead create an object type that mimics the structure of my relational table, and then define a nested table TYPE against that object type:

```
TYPE TickerType AS OBJECT (
  ticker VARCHAR2(10),
  pricedate DATE
  pricetype VARCHAR2(1),
  price NUMBER);

TYPE TickerTypeSet AS TABLE OF TickerType;
```

For my table function to stream data from one stage of transformation to the next, it will have to accept as its argument a set of data—in essence, a query. The only way to do that is to pass in a cursor variable, so I will need a REF CURSOR type to use in the parameter list of my function.

I create a package to hold the REF CURSOR type based on my new nested table type:

```
PACKAGE refcur_pkg
IS
   TYPE refcur_t IS REF CURSOR RETURN StockTable%ROWTYPE;
END refcur_pkg;
```

Finally, I can write my stock pivot function:

```
     /* File on web: tabfunc_streaming.sql */
1    FUNCTION stockpivot (dataset refcur_pkg.refcur_t)
2       RETURN tickertypeset
3    IS
4       l_row_as_object tickertype := tickertype (NULL, NULL, NULL, NULL);
5       l_row_from_query  dataset%ROWTYPE;
6       retval tickertypeset := tickertypeset ();
7    BEGIN
8       LOOP
9          FETCH dataset
10           INTO l_row_from_query;
11
12          EXIT WHEN dataset%NOTFOUND;
13          -- Create the OPEN object type instance
14          l_row_as_object.ticker := l_row_from_query.ticker;
15          l_row_as_object.pricetype := 'O';
16          l_row_as_object.price := l_row_from_query.open_price;
17          l_row_as_object.pricedate := l_row_from_query.trade_date;
18          retval.EXTEND;
19          retval (retval.LAST) := l_row_as_object;
20          -- Create the CLOSED object type instance
21          l_row_as_object.pricetype := 'C';
22          l_row_as_object.price := l_row_from_query.close_price;
23          retval.EXTEND;
24          retval (retval.LAST) := l_row_as_object;
25       END LOOP;
26
27       CLOSE dataset;
28
29       RETURN retval;
30    END stockpivot;
```

As with the pet_family function, the specifics of this program are not important, and your own transformation logic will be qualitatively more complex. The basic steps performed here, however, will likely be repeated in your own code, so I will review them in the following table.

Line(s)	Description
1–2	The function header: pass in a result set as a cursor variable, and return a nested table based on the object type.
4	Declare a local object, which will be used to populate the nested table.
5	Declare a local record based on the result set. This will be populated by the FETCH from the cursor variable.
6	The local nested table that will be returned by the function.

Line(s)	Description
8–12	Start up a simple loop to fetch each row separately from the cursor variable, terminating the loop when no more data is in the cursor.
14–19	Use the "open" data in the record to populate the local object, and then place it in the nested table, after EXTENDing to define the new row.
21–25	Use the "close" data in the record to populate the local object, and then place it in the nested table, after EXTENDing to define the new row.
27–30	Close the cursor and return the nested table. Mission completed. Really.

And now that I have this function in place to do all the fancy but necessary footwork, I can use it inside my query to stream data from one table to another:

```
BEGIN
   INSERT INTO tickertable
      SELECT *
        FROM TABLE (stockpivot (CURSOR (SELECT *
                                          FROM stocktable)));
END;
```

My inner SELECT retrieves all rows in the stocktable. The CURSOR expression around that query transforms the result set into a cursor variable, which is passed to stockpivot. That function returns a nested table, and the TABLE operator then translates it into a relational table format that can be queried.

It may not be magic, but it *is* a bit magical, wouldn't you say? Well, if you think a streaming function is special, check out pipelined functions!

Creating a pipelined function

A *pipelined function* is a table function that returns a result set as a collection but does so asynchronously to the termination of the function. In other words, the database no longer waits for the function to run to completion, storing all the rows it computes in the PL/SQL collection, before it delivers the first rows. Instead, as each row is ready to be assigned into the collection, it is piped out of the function. This section describes the basics of pipelined table functions. The performance implications of these functions are explored in detail in Chapter 21.

Let's take a look at a rewrite of the stockpivot function and see more clearly what is needed to build pipelined functions:

```
/* File on web: tabfunc_pipelined.sql */
FUNCTION stockpivot (dataset refcur_pkg.refcur_t)
RETURN tickertypeset PIPELINED
IS
   l_row_as_object tickertype := tickertype (NULL, NULL, NULL, NULL);
   l_row_from_query  dataset%ROWTYPE;
BEGIN
   LOOP
```

```
 8          FETCH dataset INTO l_row_from_query;
 9          EXIT WHEN dataset%NOTFOUND;
10
11          -- first row
12          l_row_as_object.ticker := l_row_from_query.ticker;
13          l_row_as_object.pricetype := 'O';
14          l_row_as_object.price := l_row_from_query.open_price;
15          l_row_as_object.pricedate := l_row_from_query.trade_date;
16          PIPE ROW (l_row_as_object);
17
18          -- second row
19          l_row_as_object.pricetype := 'C';
20          l_row_as_object.price := l_row_from_query.close_price;
21          PIPE ROW (l_row_as_object);
22       END LOOP;
23
24       CLOSE dataset;
25       RETURN;
26    END;
```

The following table notes several changes to our original functionality.

Line(s)	Description
2	The only change from the original stockpivot function is the addition of the PIPELINED keyword.
4–5	Declare a local object and local record, as with the first stockpivot. What's striking about these lines is what I *don't* declare—namely, the nested table that will be returned by the function. A hint of what is to come...
7–9	Start up a simple loop to fetch each row separately from the cursor variable, terminating the loop when no more data is in the cursor.
12–15 and 19–21	Populate the local object for the open and close tickertable rows to be placed in the nested table.
16 and 21	Use the PIPE ROW statement (valid only in pipelined functions) to "pipe" the objects immediately out from the function.
25	At the bottom of the executable section, the function doesn't return anything! Instead, it calls the unqualified RETURN (formerly allowed only in procedures) to return control to the calling block. The function already returned all of its data with the PIPE ROW statements.

You can call the pipelined function as you would the nonpipelined version. You won't see any difference in behavior, unless you set up the pipelined function to be executed in parallel as part of a parallel query (covered in the next section) or include logic that takes advantage of the asynchronous return of data.

Consider, for example, a query that uses the ROWNUM pseudocolumn to restrict the rows of interest:

```
BEGIN
   INSERT INTO tickertable
      SELECT *
        FROM TABLE (stockpivot (CURSOR (SELECT *
                                          FROM stocktable)))
        WHERE ROWNUM < 10;
END;
```

My tests show that on Oracle Database 10*g* and Oracle Database 11*g*, if I pivot 100,000 rows into 200,000 and then return only the first 9 rows, the pipelined version completes its work in 0.2 seconds, while the nonpipelined version takes 4.6 seconds.

Clearly, piping rows back does work and does make a noticeable difference!

Enabling a function for parallel execution

One enormous step forward for PL/SQL, introduced first in Oracle9*i* Database, is the ability to execute functions within a parallel query context. Prior to Oracle9*i* Database, a call to a PL/SQL function inside SQL caused serialization of that query—a major problem for data warehousing applications. You can now add information to the header of a pipelined function in order to instruct the runtime engine how the data set being passed into the function should be partitioned for parallel execution.

In general, if you would like your function to execute in parallel, it must have a single, strongly typed REF CURSOR input parameter.[1]

Here are some examples:

- Specify that the function can run in parallel and that the data passed to that function can be partitioned arbitrarily:

  ```
  FUNCTION my_transform_fn (
       p_input_rows in employee_info.recur_t )
    RETURN employee_info.transformed_t
    PIPELINED
    PARALLEL_ENABLE ( PARTITION p_input_rows BY ANY )
  ```

  In this example, the keyword ANY expresses the programmer's assertion that the results are independent of the order in which the function gets the input rows. When this keyword is used, the runtime system randomly partitions the data among the various query processes. This keyword is appropriate for use with functions that take in one row, manipulate its columns, and generate output rows based on the columns of this row only. If your program has other dependencies, the outcome will be unpredictable.

- Specify that the function can run in parallel, that all the rows for a given department go to the same process, and that all of these rows are delivered consecutively:

  ```
  FUNCTION my_transform_fn (
     p_input_rows in employee_info.recur_t )
  RETURN employee_info.transformed_t
  PIPELINED
  ```

1. The input REF CURSOR need *not* be strongly typed to be partitioned by ANY.

```
CLUSTER P_INPUT_ROWS BY (department)
PARALLEL_ENABLE
  ( PARTITION P_INPUT_ROWS BY HASH (department) )
```

Oracle uses the term *clustered* to signify this type of delivery, and *cluster key* for the column (in this case, department) on which the aggregation is done. But significantly, the algorithm does *not* care in what order of cluster key it receives each successive cluster, and Oracle doesn't guarantee any particular order here. This allows for a quicker algorithm than if rows were required to be clustered and delivered in the order of the cluster key. It scales as *order N* rather than *order N.log(N)*, where *N* is the number of rows.

In this example, I can choose between HASH (department) and RANGE (department), depending on what I know about the distribution of the values. HASH is quicker than RANGE and is the natural choice to be used with CLUSTER...BY.

- Specify that the function can run in parallel and that the rows that are delivered to a particular process, as directed by PARTITION...BY (for that specified partition), will be locally sorted by that process. The effect will be to parallelize the sort:

```
FUNCTION my_transform_fn (
   p_input_rows in employee_info.recur_t )
RETURN employee_info.transformed_t
PIPELINED
ORDER P_INPUT_ROWS BY (C1)
PARALLEL_ENABLE
  ( PARTITION P_INPUT_ROWS BY RANGE (C1) )
```

Because the sort is parallelized, there should be no ORDER...BY in the SELECT used to invoke the table function. (In fact, an ORDER...BY clause in the SELECT statement would subvert the attempt to parallelize the sort.) Thus, it's natural to use the RANGE option together with the ORDER...BY option. This will be slower than CLUSTER...BY, and so should be used only when the algorithm depends on it.

The CLUSTER...BY construct can't be used together with the ORDER...BY in the declaration of a table function. This means that an algorithm that depends on clustering on one key, c1, and then on ordering within the set row for a given value of c1 by, say, c2, would have to be parallelized by using the ORDER...BY in the declaration in the table function.

Deterministic Functions

A function is considered to be *deterministic* if it returns the same result value whenever it is called with the same values for its IN and IN OUT arguments. Another way to think about deterministic programs is that they have no side effects. Everything the program changes is reflected in the parameter list.

The following function (a simple encapsulation on top of SUBSTR) is a deterministic function:

```
FUNCTION betwnstr (
   string_in IN VARCHAR2, start_in IN PLS_INTEGER, end_in IN PLS_INTEGER)
   RETURN VARCHAR2 IS
BEGIN
   RETURN (SUBSTR (string_in, start_in, end_in - start_in + 1));
END betwnstr;
```

As long as I pass in, for example, "abcdef" for the string, 3 for the start, and 5 for the end, betwnStr will always return "cde". Now, if that is the case, why not have the database save the results associated with a set of arguments? Then when I next call the function with those arguments, it can return the result without executing the function!

You can achieve this effect when calling your function inside a SQL statement by adding the DETERMINISTIC clause to the function's header, as in the following:

```
FUNCTION betwnstr (
   string_in IN VARCHAR2, start_in IN PLS_INTEGER, end_in IN PLS_INTEGER)
   RETURN VARCHAR2 DETERMINISTIC
```

The decision to use a saved copy of the function's return result (if such a copy is available) is made by the Oracle query optimizer. Saved copies can come from a materialized view, a function-based index, or a repetitive call to the same function in the same SQL statement.

You must declare a function as DETERMINISTIC in order for it to be called in the expression of a function-based index, or from the query of a materialized view if that view is marked REFRESH FAST or ENABLE QUERY REWRITE. Also, deterministic caching of your function's inputs and results will occur only when the function is called inside a SQL statement.

Use of deterministic functions can improve the performance of SQL statements that call such functions. For more information on using deterministic functions as a caching mechanism, see Chapter 21. That chapter also describes the function result caching mechanism introduced in Oracle Database 11*g*, specified using RESULT_CACHE.

Oracle has no way of reliably checking to make sure that a function you declare to be deterministic actually is free of any side effects. It is up to you to use this feature responsibly. Your deterministic function should not rely on package variables, nor should it access the database in a way that might affect the result set.

For a demonstration of the effect of using a deterministic function (and its limitations), check out the *deterministic.sql* file on the book's website.

Implicit Cursor Results (Oracle Database 12c)

You know what I really hate? When a developer claims that Transact SQL is better than PL/SQL—and points out a feature that actually *is* better. Generally, I would argue that PL/SQL is head and shoulders above Transact SQL, but hey, every language has its strengths and weaknesses, and Transact SQL has long supported the ability to create a procedure that simply returns the contents of a result set to the screen. With PL/SQL, until recently you would have had to write a query, iterate through the result set, and call DBMS_OUTPUT.PUT_LINE to display the results.

However, this functionality has now been made available in PL/SQL. The addition will be of benefit primarily to developers and applications migrating from Transact SQL to PL/SQL (welcome!), and also as a testing aid (it's now much easier to write a quick procedure to verify the contents of a table).

Oracle implements this functionality by adding new functionality to the DBMS_SQL package. Hurray! Another reason to not give up on this oldie-but-goodie package (largely made irrelevant by native dynamic SQL).

Suppose I want to display the last names of all employees in a given department. I can now write the following procedure:

```
/* File on web: 12c_return_result.sql */
CREATE OR REPLACE PROCEDURE show_emps (
   department_id_in IN employees.department_id%TYPE)
IS
   l_cursor   SYS_REFCURSOR;
BEGIN
   OPEN l_cursor FOR
        SELECT last_name
          FROM employees
         WHERE department_id = department_id_in
      ORDER BY last_name;

   DBMS_SQL.return_result (l_cursor);
END;
/
```

And when I execute it in SQL*Plus for department ID 20, I see the following:

```
BEGIN
   show_emps (20);
END;
/
PL/SQL procedure successfully completed.

ResultSet #1

LAST_NAME
-------------------------
```

```
    Fay
    Hartstein
```

You can also use the DBMS_SQL.GET_NEXT_RESULT procedure to get the next cursor, which was returned with a call to RETURN_RESULT, from *inside* your PL/SQL program, rather than passing it back to the host environment.

Go Forth and Modularize!

PL/SQL has a long history of establishing the foundation of code for large and complex applications. Companies run their businesses on PL/SQL-based applications, and they use these applications for *years—even decades*. To be quite honest, you don't have much of a chance of success building (and certainly maintaining) such large-scale, mission-critical systems without an intimate familiarity with (application of) the modularization techniques available in PL/SQL.

This book should provide you with some solid pointers and a foundation on which to build your code. There is still much more for you to learn, especially with regard to the awesome range of supplied packages that Oracle Corporation provides with various tools and the database itself, such as DBMS_RLS (for row-level security) and UTL_TCP (for TCP-related functionality).

Behind all that technology, however, I strongly encourage you to develop a firm commitment to modularization and reuse. Develop a deep and abiding allergy to code redundancy and to the hardcoding of values and formulas. Apply a fanatic's devotion to the modular construction of true black boxes that easily plug and play in and across applications.

You will then find that you spend more time in the *design* phase of your development and less time debugging your code (joy of joys!). Your programs will be more readable and more maintainable. They will stand as elegant testimonies to your intellectual integrity. You will be the most popular kid in the class.