**1****Overview of PL/SQL**

PL/SQL, the Oracle procedural extension of SQL, is a completely portable, high-performance transaction-processing language. This chapter explains its advantages and briefly describes its main features and its architecture.

**Advantages of PL/SQL**

PL/SQL has these advantages:

* [Tight Integration with SQL](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/overview.htm#CJAGBBAD)
* [High Performance](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/overview.htm#CJAHAGEF)
* [High Productivity](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/overview.htm#CJABDHHJ)
* [Full Portability](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/overview.htm#CJADFBDD)
* [Tight Security](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/overview.htm#CJAFDGHC)
* [Access to Predefined Packages](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/overview.htm#CJAJDHCF)
* [Support for Object-Oriented Programming](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/overview.htm#CJAFFJDB)
* [Support for Developing Web Applications and Server Pages](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/overview.htm#CJAIEIJA)

**Tight Integration with SQL**

SQL has become the standard database language because it is flexible, powerful, and easy to learn. A few English-like statements such asSELECT, INSERT, UPDATE, and DELETE make it easy to manipulate the data stored in a relational database.

PL/SQL is tightly integrated with SQL. With PL/SQL, you can use all SQL data manipulation, cursor control, and transaction control statements, and all SQL functions, operators, and pseudocolumns.

PL/SQL fully supports SQL data types. You need not convert between PL/SQL and SQL data types. For example, if your PL/SQL program retrieves a value from a database column of the SQL type VARCHAR2, it can store that value in a PL/SQL variable of the type VARCHAR2. Special PL/SQL language features let you work with table columns and rows without specifying the data types, saving on maintenance work when the table definitions change.

Running a SQL query and processing the result set is as easy in PL/SQL as opening a text file and processing each line in popular scripting languages. Using PL/SQL to access metadata about database objects and handle database error conditions, you can write utility programs for database administration that are reliable and produce readable output about the success of each operation. Many database features, such as triggers and object types, use PL/SQL. You can write the bodies of triggers and methods for object types in PL/SQL.

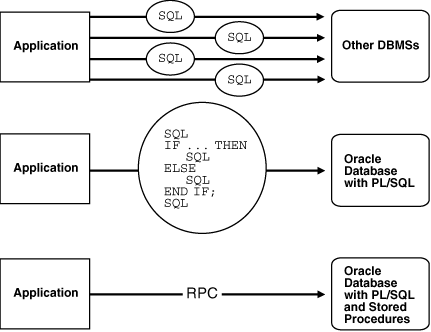
PL/SQL supports both static and dynamic SQL. **Static SQL** is SQL whose full text is known at compilation time. **Dynamic SQL** is SQL whose full text is not known until run time. Dynamic SQL enables you to make your applications more flexible and versatile. For information about using static SQL with PL/SQL, see [Chapter 6, "Using Static SQL."](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/static.htm#BABGEDAE) For information about using dynamic SQL, see [Chapter 7, "Using Dynamic SQL."](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/dynamic.htm#CACDDACH)

**High Performance**

With PL/SQL, an entire block of statements can be sent to the database at one time. This can drastically reduce network traffic between the database and an application. As [Figure 1-1](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/overview.htm#i8981) shows, you can use PL/SQL blocks and subprograms (procedures and functions) to group SQL statements before sending them to the database for execution. PL/SQL also has language features to further speed up SQL statements that are issued inside a loop.

PL/SQL stored subprograms are compiled once and stored in executable form, so subprogram calls are efficient. Because stored subprograms execute in the database server, a single call over the network can start a large job. This division of work reduces network traffic and improves response times. Stored subprograms are cached and shared among users, which lowers memory requirements and call overhead.

***Figure 1-1 PL/SQL Boosts Performance***

  
[Description of "Figure 1-1 PL/SQL Boosts Performance"](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/img_text/lnpls005.htm)

**High Productivity**

PL/SQL lets you write very compact code for manipulating data. In the same way that scripting languages such as PERL can read, transform, and write data from files, PL/SQL can query, transform, and update data in a database. PL/SQL saves time on design and debugging by offering a full range of software-engineering features, such as exception handling, encapsulation, data hiding, and object-oriented data types.

PL/SQL extends tools such as Oracle Forms. With PL/SQL in these tools, you can use familiar language constructs to build applications. For example, you can use an entire PL/SQL block in an Oracle Forms trigger, instead of multiple trigger steps, macros, or user exits. PL/SQL is the same in all environments. After you learn PL/SQL with one Oracle tool, you can transfer your knowledge to other tools.

**Full Portability**

Applications written in PL/SQL can run on any operating system and platform where the database runs. With PL/SQL, you can write portable program libraries and reuse them in different environments.

**Tight Security**

PL/SQL stored subprograms move application code from the client to the server, where you can protect it from tampering, hide the internal details, and restrict who has access. For example, you can grant users access to a subprogram that updates a table, but not grant them access to the table itself or to the text of the UPDATE statement. Triggers written in PL/SQL can control or record changes to data, making sure that all changes obey your business rules.

For information about wrapping, or hiding, the source of a PL/SQL unit, see [Appendix A, "Wrapping PL/SQL Source Code"](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/wrap.htm#BABEFEJF).

**Access to Predefined Packages**

Oracle provides product-specific packages that define APIs you can invoke from PL/SQL to perform many useful tasks. These packages include DBMS\_ALERT for using triggers, DBMS\_FILE for reading and writing operating system text files, UTL\_HTTP for making hypertext transfer protocol (HTTP) callouts, DBMS\_OUTPUT for display output from PL/SQL blocks and subprograms, and DBMS\_PIPE for communicating over named pipes. For more information about these packages, see [Overview of Product-Specific PL/SQL Packages](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/packages.htm#i2432).

For complete information about the packages supplied by Oracle, see [*Oracle Database PL/SQL Packages and Types Reference*](http://docs.oracle.com/cd/B28359_01/appdev.111/b28419/toc.htm).

**Support for Object-Oriented Programming**

Object types are an ideal object-oriented modeling tool, which you can use to reduce the cost and time required to build complex applications. Besides enabling you to create software components that are modular, maintainable, and reusable, object types allow different teams of programmers to develop software components concurrently.

By encapsulating operations with data, object types let you move data-maintenance code out of SQL scripts and PL/SQL blocks into methods. Also, object types hide implementation details, so that you can change the details without affecting client programs.

In addition, object types allow for realistic data modeling. Complex real-world entities and relationships map directly into object types. This direct mapping helps your programs better reflect the world they are trying to simulate. For information about object types, see [*Oracle Database Object-Relational Developer's Guide*](http://docs.oracle.com/cd/B28359_01/appdev.111/b28371/adobjint.htm#ADOBJ001).

**Support for Developing Web Applications and Server Pages**

You can use PL/SQL to develop Web applications and Server Pages (PSPs). For more information, see [Using PL/SQL to Create Web Applications](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/fundamentals.htm#CIHDJJDI) and [Using PL/SQL to Create Server Pages](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/fundamentals.htm#CIHFHCCA).

**Main Features of PL/SQL**

PL/SQL combines the data-manipulating power of SQL with the processing power of procedural languages.

When a problem can be solved using SQL, you can issue SQL statements from your PL/SQL programs, without learning new APIs.

Like other procedural programming languages, PL/SQL lets you declare constants and variables, control program flow, define subprograms, and trap run-time errors.

You can break complex problems into easily understandable subprograms, which you can reuse in multiple applications.

Topics:

* [PL/SQL Blocks](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/overview.htm#i8859)
* [PL/SQL Error Handling](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/overview.htm#g12979)
* [PL/SQL Input and Output](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/overview.htm#CBJGHHBG)
* [PL/SQL Variables and Constants](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/overview.htm#g12828)
* [PL/SQL Data Abstraction](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/overview.htm#g12947)
* [PL/SQL Control Structures](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/overview.htm#i7369)
* [PL/SQL Subprograms](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/overview.htm#g1461293)
* [PL/SQL Packages (APIs Written in PL/SQL)](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/overview.htm#BABFHIEE)
* [Conditional Compilation](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/overview.htm#g1115615)
* [Embedded SQL Statements](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/overview.htm#g12862)

**PL/SQL Blocks**

The basic unit of a PL/SQL source program is the block, which groups related declarations and statements.

A PL/SQL block is defined by the keywords DECLARE, BEGIN, EXCEPTION, and END. These keywords partition the block into a declarative part, an executable part, and an exception-handling part. Only the executable part is required.

Declarations are local to the block and cease to exist when the block completes execution, helping to avoid cluttered namespaces for variables and subprograms.

Blocks can be nested: Because a block is an executable statement, it can appear in another block wherever an executable statement is allowed.

[Example 1-1](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/overview.htm#BABDHBBA) shows the basic structure of a PL/SQL block. For the formal syntax description, see [Block](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/block.htm#i32791).

***Example 1-1 PL/SQL Block Structure***

DECLARE -- Declarative part (optional)

-- Declarations of local types, variables, & subprograms

BEGIN -- Executable part (required)

-- Statements (which can use items declared in declarative part)

[EXCEPTION -- Exception-handling part (optional)

-- Exception handlers for exceptions raised in executable part]

END;

A PL/SQL block can be submitted to an interactive tool (such as SQL\*Plus or Enterprise Manager) or embedded in an Oracle Precompiler or OCI program. The interactive tool or program executes the block only once. The block is not stored in the database.

A named PL/SQL block—a subprogram—can be invoked repeatedly (see [PL/SQL Subprograms](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/overview.htm#g1461293)).

**Note:**

A block that is not stored in the database is called an **anonymous block**, even if it has a label.

**PL/SQL Error Handling**

PL/SQL makes it easy to detect and process error conditions, which are called **exceptions**. When an error occurs, an exception is raised: normal execution stops and control transfers to special exception-handling code, which comes at the end of any PL/SQL block. Each different exception is processed by a particular exception handler.

PL/SQL exception handling differs from the manual checking that you do in C programming, where you insert a check to make sure that every operation succeeded. Instead, the checks and calls to error routines are performed automatically, similar to the exception mechanism in Java programming.

Predefined exceptions are raised automatically for certain common error conditions involving variables or database operations. For example, if you try to divide a number by zero, PL/SQL raises the predefined exception ZERO\_DIVIDE automatically.

You can define exceptions of your own, for conditions that you decide are errors, or to correspond to database errors that normally result in ORA-*n* error messages. When you detect a user-defined error condition, you raise an exception with either a RAISE statement or the procedure DBMS\_STANDARD.RAISE\_APPLICATION\_ERROR. See the exception comm\_missing in [Example 1-16](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/overview.htm#BABBHHAI). In the example, if the commission is null, the exception comm\_missing is raised.

Typically, you put an exception handler at the end of a subprogram to handle exceptions that are raised anywhere inside the subprogram. To continue executing from the spot where an exception happens, enclose the code that might raise an exception inside another BEGIN-END block with its own exception handler. For example, you might put separate BEGIN-END blocks around groups of SQL statements that might raise NO\_DATA\_FOUND, or around arithmetic operations that might raise DIVIDE\_BY\_ZERO. By putting a BEGIN-END block with an exception handler inside a loop, you can continue executing the loop even if some loop iterations raise exceptions. See [Example 5-38](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/collections.htm#BABFJEIA).

For information about PL/SQL errors, see [Overview of PL/SQL Run-Time Error Handling](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/errors.htm#i3336). For information about PL/SQL warnings, see[Overview of PL/SQL Compile-Time Warnings](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/errors.htm#i9084).

**PL/SQL Input and Output**

Most PL/SQL input and output (I/O) is through SQL statements that store data in database tables or query those tables. All other PL/SQL I/O is done through APIs, such as the PL/SQL package DBMS\_OUTPUT.

To display output passed to DBMS\_OUTPUT, you need another program, such as SQL\*Plus. To see DBMS\_OUTPUT output with SQL\*Plus, you must first issue the SQL\*Plus command SET SERVEROUTPUT ON. For information about SET SERVEROUTPUT ON, see [*SQL\*Plus User's Guide and Reference*](http://docs.oracle.com/cd/B28359_01/server.111/b31189/ch12040.htm#SQPUG099).

Other PL/SQL APIs for processing I/O are provided by packages such as:

| **Package(s)** | **PL/SQL uses package ...** |
| --- | --- |
| HTF and HTP | to display output on a web page |
| DBMS\_PIPE | to pass information between PL/SQL and operating-system commands |
| UTL\_FILE | to reads and write operating system files |
| UTL\_HTTP | to communicate with web servers |
| UTL\_SMTP | to communicate with mail servers |

Although some of the preceding APIs can accept input as well as display output, they have cannot accept data directly from the keyboard. For that, use the SQL\*Plus commands PROMPT and ACCEPT.

**PL/SQL Variables and Constants**

PL/SQL lets you declare variables and constants, and then use them in SQL and procedural statements anywhere an expression can be used. You must declare a variable or constant before referencing it in any other statements. For more information, see [Declarations](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/fundamentals.htm#i27306).

Topics:

* [Declaring PL/SQL Variables](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/overview.htm#CJAFEIFF)
* [Assigning Values to Variables](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/overview.htm#g12589)
* [Declaring PL/SQL Constants](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/overview.htm#g12604)
* [Bind Variables](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/overview.htm#g1168839)

**Declaring PL/SQL Variables**

A PL/SQL variable can have any SQL data type (such as CHAR, DATE, or NUMBER) or a PL/SQL-only data type (such as BOOLEAN orPLS\_INTEGER).

[Example 1-2](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/overview.htm#BABEABGF) declares several PL/SQL variables. One has a PL/SQL-only data type; the others have SQL data types.

***Example 1-2 PL/SQL Variable Declarations***

SQL> DECLARE

2 part\_number NUMBER(6); -- SQL data type

3 part\_name VARCHAR2(20); -- SQL data type

4 in\_stock BOOLEAN; -- PL/SQL-only data type

5 part\_price NUMBER(6,2); -- SQL data type

6 part\_description VARCHAR2(50); -- SQL data type

7 BEGIN

8 NULL;

9 END;

10 /

PL/SQL procedure successfully completed.

SQL>

For more information about PL/SQL data types, see [Chapter 3, "PL/SQL Data Types."](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/datatypes.htm#CJAEDAEA)

PL/SQL also lets you declare composite data types, such as nested tables, variable-size arrays, and records. For more informations, see[Chapter 5, "Using PL/SQL Collections and Records."](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/collections.htm#CIHIEBJC)

**Assigning Values to Variables**

You can assign a value to a variable in the following ways:

* With the assignment operator (:=), as in [Example 1-3](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/overview.htm#BABBDCEC).
* By selecting (or fetching) database values into it, as in [Example 1-4](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/overview.htm#BABHFEHA).
* By passing it as an OUT or IN OUT parameter to a subprogram, and then assigning the value inside the subprogram, as in [Example 1-5](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/overview.htm#BABIIBDF)

***Example 1-3 Assigning Values to Variables with the Assignment Operator***

SQL> DECLARE -- You can assign values here

2 wages NUMBER;

3 hours\_worked NUMBER := 40;

4 hourly\_salary NUMBER := 22.50;

5 bonus NUMBER := 150;

6 country VARCHAR2(128);

7 counter NUMBER := 0;

8 done BOOLEAN;

9 valid\_id BOOLEAN;

10 emp\_rec1 employees%ROWTYPE;

11 emp\_rec2 employees%ROWTYPE;

12 TYPE commissions IS TABLE OF NUMBER INDEX BY PLS\_INTEGER;

13 comm\_tab commissions;

14

15 BEGIN -- You can assign values here too

16 wages := (hours\_worked \* hourly\_salary) + bonus;

17 country := 'France';

18 country := UPPER('Canada');

19 done := (counter > 100);

20 valid\_id := TRUE;

21 emp\_rec1.first\_name := 'Antonio';

22 emp\_rec1.last\_name := 'Ortiz';

23 emp\_rec1 := emp\_rec2;

24 comm\_tab(5) := 20000 \* 0.15;

25 END;

26 /

PL/SQL procedure successfully completed.

SQL>

In [Example 1-4](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/overview.htm#BABHFEHA), 10% of an employee's salary is selected into the bonus variable. Now you can use the bonus variable in another computation or insert its value into a database table.

***Example 1-4 Using SELECT INTO to Assign Values to Variables***

SQL> DECLARE

2 bonus NUMBER(8,2);

3 emp\_id NUMBER(6) := 100;

4 BEGIN

5 **SELECT salary \* 0.10 INTO bonus**

6 **FROM employees**

7 **WHERE employee\_id = emp\_id;**

8 END;

9 /

PL/SQL procedure successfully completed.

SQL>

[Example 1-5](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/overview.htm" \l "BABIIBDF) passes the new\_sal variable to a subprogram, and the subprogram updates the variable.

***Example 1-5 Assigning Values to Variables as Parameters of a Subprogram***

SQL> DECLARE

2 **new\_sal NUMBER(8,2);**

3 emp\_id NUMBER(6) := 126;

4

5 PROCEDURE adjust\_salary (

6 emp\_id NUMBER,

7 **sal IN OUT NUMBER**

8 ) IS

9 emp\_job VARCHAR2(10);

10 avg\_sal NUMBER(8,2);

11 BEGIN

12 SELECT job\_id INTO emp\_job

13 FROM employees

14 WHERE employee\_id = emp\_id;

15

16 SELECT AVG(salary) INTO avg\_sal

17 FROM employees

18 WHERE job\_id = emp\_job;

19

20 DBMS\_OUTPUT.PUT\_LINE ('The average salary for '

21 || emp\_job

22 || ' employees: '

23 || TO\_CHAR(avg\_sal)

24 );

25

26 **sal := (sal + avg\_sal)/2;**

27 END;

28

29 BEGIN

30 SELECT AVG(salary) INTO new\_sal

31 FROM employees;

32

33 DBMS\_OUTPUT.PUT\_LINE ('The average salary for all employees: '

34 || TO\_CHAR(new\_sal)

35 );

36

37 **adjust\_salary(emp\_id, new\_sal);**

38 END;

39 /

The average salary for all employees: 6461.68

The average salary for ST\_CLERK employees: 2785

PL/SQL procedure successfully completed.

SQL>

**Declaring PL/SQL Constants**

Declaring a PL/SQL constant is like declaring a PL/SQL variable except that you must add the keyword CONSTANT and immediately assign a value to the constant. For example:

credit\_limit CONSTANT NUMBER := 5000.00;

No further assignments to the constant are allowed.

**Bind Variables**

Bind variables improve performance by allowing the database to reuse SQL statements.

When you embed a SQL INSERT, UPDATE, DELETE, or SELECT statement directly in your PL/SQL code, PL/SQL turns the variables in the WHEREand VALUES clauses into bind variables automatically. The database can reuse these SQL statements each time the same code is executed. To run similar statements with different variable values, you can save parsing overhead by invoking a stored subprogram that accepts parameters and then issues the statements with the parameters substituted in the appropriate places.

PL/SQL does not create bind variables automatically when you use dynamic SQL, but you can use them with dynamic SQL by specifying them explicitly.

**PL/SQL Data Abstraction**

Data abstraction lets you work with the essential properties of data without being too involved with details. After you design a data structure, you can focus on designing algorithms that manipulate the data structure.

Topics:

* [Cursors](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/overview.htm#BABHIDEG)
* [%TYPE Attribute](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/overview.htm#g12611)
* [%ROWTYPE Attribute](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/overview.htm#g12629)
* [Collections](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/overview.htm#g12698)
* [Records](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/overview.htm#g12708)
* [Object Types](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/overview.htm#BABJCCAI)

**Cursors**

A cursor is a name for a specific private SQL area in which information for processing the specific statement is kept. PL/SQL uses both implicit and explicit cursors. PL/SQL implicitly declares a cursor for all SQL data manipulation statements on a set of rows, including queries that return only one row. For queries that return more than one row, you can explicitly declare a cursor to process the rows individually. For example, [Example 1-6](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/overview.htm#BABFAAAJ) declares an explicit cursor.

For information about cursors, see [Managing Cursors in PL/SQL](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/static.htm#i45288).

**%TYPE Attribute**

The %TYPE attribute provides the data type of a variable or database column. This is particularly useful when declaring variables that will hold database values. For example, assume there is a column named last\_name in a table named employees. To declare a variable namedv\_last\_name that has the same data type as column last\_name, use dot notation and the %TYPE attribute, as follows:

v\_last\_name employees.last\_name%TYPE;

Declaring v\_last\_name with %TYPE has two advantages. First, you need not know the exact data type of last\_name. Second, if you change the database definition of last\_name, perhaps to make it a longer character string, the data type of v\_last\_name changes accordingly at run time.

For more information about %TYPE, see [Using the %TYPE Attribute](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/fundamentals.htm#i6080) and [%TYPE Attribute](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/type_attribute.htm#i36397).

**%ROWTYPE Attribute**

In PL/SQL, records are used to group data. A record consists of a number of related fields in which data values can be stored. The%ROWTYPE attribute provides a record type that represents a row in a table. The record can store an entire row of data selected from the table or fetched from a cursor or cursor variable. See [Cursors](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/overview.htm#BABHIDEG).

Columns in a row and corresponding fields in a record have the same names and data types. In the following example, you declare a record named dept\_rec, whose fields have the same names and data types as the columns in the departments table:

dept\_rec departments%ROWTYPE; -- declare record variable

You use dot notation to reference fields, as follows:

v\_deptid := dept\_rec.department\_id;

If you declare a cursor that retrieves the last name, salary, hire date, and job class of an employee, you can use %ROWTYPE to declare a record that stores the same information.

The FETCH statement in [Example 1-6](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/overview.htm#BABFAAAJ) assigns the value in the last\_name column of the employees table to the last\_name field ofemployee\_rec, the value in the salary column is to the salary field, and so on.

***Example 1-6 Using %ROWTYPE with an Explicit Cursor***

SQL> DECLARE

2 CURSOR **c1** IS

3 SELECT last\_name, salary, hire\_date, job\_id

4 FROM employees

5 WHERE employee\_id = 120;

6

7 **employee\_rec c1%ROWTYPE;**

8

9 BEGIN

10 OPEN c1;

11 **FETCH c1 INTO employee\_rec;**

12 DBMS\_OUTPUT.PUT\_LINE('Employee name: ' || employee\_rec.last\_name);

13 END;

14 /

Employee name: Weiss

PL/SQL procedure successfully completed.

SQL>

For more information about %ROWTYPE, see [Using the %ROWTYPE Attribute](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/fundamentals.htm#i6079) and [%ROWTYPE Attribute](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/rowtype_attribute.htm#i35991).

**Collections**

PL/SQL collection types let you declare high-level data types similar to arrays, sets, and hash tables found in other languages. In PL/SQL, array types are known as varrays (short for variable-size arrays), set types are known as nested tables, and hash table types are known as associative arrays. Each kind of collection is an ordered group of elements, all of the same type. Each element has a unique subscript that determines its position in the collection. When declaring collections, you use a TYPE definition. See [Defining Collection Types](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/collections.htm#i20383).

To reference an element, use subscript notation with parentheses, as shown in [Example 1-7](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/overview.htm#BABIHGGF).

***Example 1-7 Using a PL/SQL Collection Type***

SQL> DECLARE

2 **TYPE staff\_list IS TABLE OF employees.employee\_id%TYPE;**

3 **staff staff\_list;**

4 lname employees.last\_name%TYPE;

5 fname employees.first\_name%TYPE;

6 BEGIN

7 **staff := staff\_list(100, 114, 115, 120, 122);**

8

9 FOR i IN staff.FIRST..staff.LAST LOOP

10 SELECT last\_name, first\_name INTO lname, fname

11 FROM employees

12 WHERE employees.employee\_id = **staff(i)**;

13

14 DBMS\_OUTPUT.PUT\_LINE (TO\_CHAR(**staff(i)**)

15 || ': '

16 || lname

17 || ', '

18 || fname

19 );

20 END LOOP;

21 END;

22 /

100: King, Steven

114: Raphaely, Den

115: Khoo, Alexander

120: Weiss, Matthew

122: Kaufling, Payam

PL/SQL procedure successfully completed.

SQL>

Collections can be passed as parameters, so that subprograms can process arbitrary numbers of elements.You can use collections to move data into and out of database tables using high-performance language features known as bulk SQL.

For information about collections, see [Chapter 5, "Using PL/SQL Collections and Records."](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/collections.htm#CIHIEBJC)

**Records**

Records are composite data structures whose fields can have different data types. You can use records to hold related items and pass them to subprograms with a single parameter. When declaring records, you use a TYPE definition, as in [Example 1-8](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/overview.htm#BABEGBEH). See [Defining and Declaring Records](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/collections.htm#i20479).

***Example 1-8 Declaring a Record Type***

SQL> DECLARE

2 TYPE timerec IS RECORD (

3 hours SMALLINT,

4 minutes SMALLINT

5 );

6

7 TYPE meeting\_type IS RECORD (

8 date\_held DATE,

9 duration timerec, -- nested record

10 location VARCHAR2(20),

11 purpose VARCHAR2(50)

12 );

13

14 BEGIN

15 NULL;

16 END;

17 /

PL/SQL procedure successfully completed.

SQL>

You can use the %ROWTYPE attribute to declare a record that represents a row in a table or a row from a query result set, without specifying the names and types for the fields.

For information about records, see [Chapter 5, "Using PL/SQL Collections and Records."](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/collections.htm#CIHIEBJC)

**Object Types**

PL/SQL supports object-oriented programming through object types. An object type encapsulates a data structure along with the subprograms needed to manipulate the data. The variables that form the data structure are known as attributes. The subprograms that manipulate the attributes are known as methods.

Object types reduce complexity by breaking down a large system into logical entities. This lets you create software components that are modular, maintainable, and reusable. Object-type definitions, and the code for the methods, are stored in the database. Instances of these object types can be stored in tables or used as variables inside PL/SQL code. [Example 1-9](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/overview.htm#BABJHAAH) shows an object type definition for a bank account.

***Example 1-9 Defining an Object Type***

SQL> CREATE TYPE bank\_account AS OBJECT (

2 acct\_number NUMBER(5),

3 balance NUMBER,

4 status VARCHAR2(10),

5

6 MEMBER PROCEDURE open

7 (SELF IN OUT NOCOPY bank\_account,

8 amount IN NUMBER),

9

10 MEMBER PROCEDURE close

11 (SELF IN OUT NOCOPY bank\_account,

12 num IN NUMBER,

13 amount OUT NUMBER),

14

15 MEMBER PROCEDURE deposit

16 (SELF IN OUT NOCOPY bank\_account,

17 num IN NUMBER,

18 amount IN NUMBER),

19

20 MEMBER PROCEDURE withdraw

21 (SELF IN OUT NOCOPY bank\_account,

22 num IN NUMBER,

23 amount IN NUMBER),

24

25 MEMBER FUNCTION curr\_bal (num IN NUMBER) RETURN NUMBER

26 );

27 /

Type created.

SQL>

For information about object types, see [*Oracle Database Object-Relational Developer's Guide*](http://docs.oracle.com/cd/B28359_01/appdev.111/b28371/adobjint.htm#ADOBJ001).

**PL/SQL Control Structures**

Control structures are the most important PL/SQL extension to SQL. Not only does PL/SQL let you manipulate database data, it lets you process the data using flow-of-control statements.

Topics:

* [Conditional Control](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/overview.htm#i1736)
* [Iterative Control](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/overview.htm#i8422)
* [Sequential Control](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/overview.htm#g12675)

For more information, see [Chapter 4, "Using PL/SQL Control Structures."](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/controlstructures.htm#CIHIAGAF)

**Conditional Control**

Often, it is necessary to take alternative actions depending on circumstances. The IF-THEN-ELSE statement lets you execute a sequence of statements conditionally. The IF clause checks a condition, the THEN clause defines what to do if the condition is true and the ELSE clause defines what to do if the condition is false or null. [Example 1-10](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/overview.htm#BABCBICG) shows the use of IF-THEN-ELSE to determine the salary raise an employee receives based on the current salary of the employee.

To choose among several values or courses of action, you can use CASE constructs. The CASE expression evaluates a condition and returns a value for each case. The case statement evaluates a condition and performs an action for each case, as in [Example 1-10](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/overview.htm#BABCBICG).

***Example 1-10 Using the IF-THEN-ELSE and CASE Statement for Conditional Control***

SQL> DECLARE

2 jobid employees.job\_id%TYPE;

3 empid employees.employee\_id%TYPE := 115;

4 sal employees.salary%TYPE;

5 sal\_raise NUMBER(3,2);

6 BEGIN

7 SELECT job\_id, salary INTO jobid, sal

8 FROM employees

9 WHERE employee\_id = empid;

10

11 CASE

12 WHEN jobid = 'PU\_CLERK' THEN

13 IF sal < 3000 THEN

14 sal\_raise := .12;

15 ELSE

16 sal\_raise := .09;

17 END IF;

18

19 WHEN jobid = 'SH\_CLERK' THEN

20 IF sal < 4000 THEN

21 sal\_raise := .11;

22 ELSE

23 sal\_raise := .08;

24 END IF;

25

26 WHEN jobid = 'ST\_CLERK' THEN

27 IF sal < 3500 THEN

28 sal\_raise := .10;

29 ELSE

30 sal\_raise := .07;

31 END IF;

32

33 ELSE

34 BEGIN

35 DBMS\_OUTPUT.PUT\_LINE('No raise for this job: ' || jobid);

36 END;

37 END CASE;

38

39 UPDATE employees

40 SET salary = salary + salary \* sal\_raise

41 WHERE employee\_id = empid;

42 END;

43 /

PL/SQL procedure successfully completed.

SQL>

A sequence of statements that uses query results to select alternative actions is common in database applications. Another common sequence inserts or deletes a row only if an associated entry is found in another table. You can bundle these common sequences into a PL/SQL block using conditional logic.

**Iterative Control**

LOOP statements let you execute a sequence of statements multiple times. You place the keyword LOOP before the first statement in the sequence and the keywords END LOOP after the last statement in the sequence. The following example shows the simplest kind of loop, which repeats a sequence of statements continually:

LOOP

-- sequence of statements

END LOOP;

The FOR-LOOP statement lets you specify a range of integers, then execute a sequence of statements once for each integer in the range. In[Example 1-11](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/overview.htm#BABGFHBG) the loop inserts 100 numbers, square roots, squares, and the sum of squares into a database table.

***Example 1-11 Using the FOR-LOOP***

SQL> CREATE TABLE sqr\_root\_sum (

2 num NUMBER,

3 sq\_root NUMBER(6,2),

4 sqr NUMBER,

5 sum\_sqrs NUMBER

6 );

Table created.

SQL>

SQL> DECLARE

2 s PLS\_INTEGER;

3 BEGIN

4 **FOR i in 1..100 LOOP**

5 s := (i \* (i + 1) \* (2\*i +1)) / 6; -- sum of squares

6

7 INSERT INTO sqr\_root\_sum

8 VALUES (i, SQRT(i), i\*i, s );

9 **END LOOP;**

10 END;

11 /

PL/SQL procedure successfully completed.

SQL>

The WHILE-LOOP statement associates a condition with a sequence of statements. Before each iteration of the loop, the condition is evaluated. If the condition is true, the sequence of statements is executed, then control resumes at the top of the loop. If the condition is false or null, the loop is bypassed and control passes to the next statement.

In [Example 1-12](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/overview.htm#BABFAHFH), you find the first employee who has a salary over $15000 and is higher in the chain of command than employee 120.

***Example 1-12 Using WHILE-LOOP for Control***

SQL> CREATE TABLE temp (

2 tempid NUMBER(6),

3 tempsal NUMBER(8,2),

4 tempname VARCHAR2(25)

5 );

Table created.

SQL>

SQL> DECLARE

2 sal employees.salary%TYPE := 0;

3 mgr\_id employees.manager\_id%TYPE;

4 lname employees.last\_name%TYPE;

5 starting\_empid employees.employee\_id%TYPE := 120;

6

7 BEGIN

8 SELECT manager\_id INTO mgr\_id

9 FROM employees

10 WHERE employee\_id = starting\_empid;

11

12 **WHILE sal <= 15000 LOOP**

13 SELECT salary, manager\_id, last\_name INTO sal, mgr\_id, lname

14 FROM employees

15 WHERE employee\_id = mgr\_id;

16 **END LOOP;**

17

18 INSERT INTO temp

19 VALUES (NULL, sal, lname);

20

21 EXCEPTION

22 WHEN NO\_DATA\_FOUND THEN

23 INSERT INTO temp VALUES (NULL, NULL, 'Not found');

24 END;

25 /

PL/SQL procedure successfully completed.

SQL>

The EXIT-WHEN statement lets you complete a loop if further processing is impossible or undesirable. When the EXIT statement is encountered, the condition in the WHEN clause is evaluated. If the condition is true, the loop completes and control passes to the next statement. In [Example 1-13](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/overview.htm#BABBCHGE), the loop completes when the value of total exceeds 25,000:

Similarly, the CONTINUE-WHEN statement immediately transfers control to the next iteration of the loop when there is no need to continue working on this iteration.

***Example 1-13 Using the EXIT-WHEN Statement***

SQL> CREATE TABLE temp (

2 tempid NUMBER(6),

3 tempsal NUMBER(8,2),

4 tempname VARCHAR2(25)

5 );

Table created.

SQL>

SQL> DECLARE

2 total NUMBER(9) := 0;

3 counter NUMBER(6) := 0;

4 BEGIN

5 **LOOP**

6 counter := counter + 1;

7 total := total + counter \* counter;

8 **EXIT WHEN total > 25000;**

9 **END LOOP;**

10

11 DBMS\_OUTPUT.PUT\_LINE ('Counter: '

12 || TO\_CHAR(counter)

13 || ' Total: '

14 || TO\_CHAR(total)

15 );

16 END;

17 /

Counter: 42 Total: 25585

PL/SQL procedure successfully completed.

SQL>

**Sequential Control**

The GOTO statement lets you branch to a label unconditionally. The label, an undeclared identifier enclosed by double angle brackets, must precede an executable statement or a PL/SQL block. When executed, the GOTO statement transfers control to the labeled statement or block, as in [Example 1-14](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/overview.htm#BABCFHBJ).

***Example 1-14 Using the GOTO Statement***

SQL> DECLARE

2 total NUMBER(9) := 0;

3 counter NUMBER(6) := 0;

4 BEGIN

5 **<<calc\_total>>**

6 counter := counter + 1;

7 total := total + counter \* counter;

8

9 IF total > 25000 THEN

10 **GOTO print\_total;**

11 ELSE

12 **GOTO calc\_total;**

13 END IF;

14

15 **<<print\_total>>**

16 DBMS\_OUTPUT.PUT\_LINE

17 ('Counter: ' || TO\_CHAR(counter) || ' Total: ' || TO\_CHAR(total));

18 END;

19 /

Counter: 42 Total: 25585

PL/SQL procedure successfully completed.

SQL>

**PL/SQL Subprograms**

A PL/SQL subprogram is a named PL/SQL block that can be invoked with a set of parameters, like double in [Example 1-15](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/overview.htm#BABHBEEC). PL/SQL has two types of subprograms, procedures and functions. A function returns a result.

***Example 1-15 PL/SQL Procedure***

SQL> DECLARE

2 in\_string VARCHAR2(100) := 'Test string';

3 out\_string VARCHAR2(200);

4

5 **PROCEDURE double (**

6 **original IN VARCHAR2,**

7 **new\_string OUT VARCHAR2**

8 **) AS**

9 **BEGIN**

10 **new\_string := original || original;**

11 **END;**

12

13 BEGIN

14 DBMS\_OUTPUT.PUT\_LINE ('in\_string: ' || in\_string);

15 **double (in\_string, out\_string);**

16 DBMS\_OUTPUT.PUT\_LINE ('out\_string: ' || out\_string);

17 END;

18 /

in\_string: Test string

out\_string: Test stringTest string

PL/SQL procedure successfully completed.

SQL>

Topics:

* [Standalone PL/SQL Subprograms](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/overview.htm#CJACCHHA)
* [Triggers](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/overview.htm#g12758)

For more information about PL/SQL subprograms, see [Chapter 8, "Using PL/SQL Subprograms."](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/subprograms.htm#CHDBEJGF)

**Standalone PL/SQL Subprograms**

You create standalone subprograms at schema level with the SQL statements CREATE PROCEDURE and CREATE FUNCTION. They are compiled and stored in the database, where they can be used by any number of applications connected to the database. When invoked, they are loaded and processed immediately. Subprograms use shared memory, so that only one copy of a subprogram is loaded into memory for execution by multiple users.

[Example 1-16](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/overview.htm#BABBHHAI) creates a standalone procedure that accepts an employee ID and a bonus amount, uses the ID to select the employee's commission percentage from a database table and to convert the commission percentage to a decimal amount, and then checks the commission amount. If the commission is null, the procedure raises an exception; otherwise, it updates the employee's salary.

***Example 1-16 Creating a Standalone PL/SQL Procedure***

SQL> CREATE OR REPLACE PROCEDURE award\_bonus (

2 emp\_id NUMBER, bonus NUMBER) AS

3 commission REAL;

4 comm\_missing EXCEPTION;

5 BEGIN

6 SELECT commission\_pct / 100 INTO commission

7 FROM employees

8 WHERE employee\_id = emp\_id;

9

10 IF commission IS NULL THEN

11 RAISE comm\_missing;

12 ELSE

13 UPDATE employees

14 SET salary = salary + bonus\*commission

15 WHERE employee\_id = emp\_id;

16 END IF;

17 EXCEPTION

18 WHEN comm\_missing THEN

19 DBMS\_OUTPUT.PUT\_LINE

20 ('This employee does not receive a commission.');

21 commission := 0;

22 WHEN OTHERS THEN

23 NULL;

24 END award\_bonus;

25 /

Procedure created.

SQL>

A PL/SQL subprogram can be invoked from an interactive tool (such as SQL\*Plus or Enterprise Manager), from an Oracle Precompiler or OCI program, from another PL/SQL subprogram, or from a trigger.

For information, about the CREATE PROCEDURE statment, see [CREATE PROCEDURE Statement](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/create_procedure.htm#i2072424).

For more information about the SQL CREATE FUNCTION, see [CREATE FUNCTION Statement](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/create_function.htm#i2153260).

[Example 1-17](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/overview.htm#BABBEFIG) invokes the stored subprogram in [Example 1-16](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/overview.htm#BABBHHAI) with the CALL statement and then from inside a block.

***Example 1-17 Invoking a Standalone Procedure from SQL\*Plus***

SQL> -- Invoke standalone procedure with CALL statement

SQL>

SQL> CALL award\_bonus(179, 1000);

Call completed.

SQL>

SQL> -- Invoke standalone procedure from within block

SQL>

SQL> BEGIN

2 award\_bonus(179, 10000);

3 END;

4 /

PL/SQL procedure successfully completed.

SQL>

Using the BEGIN-END block is recommended in several situations. For example, using the CALL statement can suppress an ORA-*n* error that was not handled in the PL/SQL subprogram.

For additional examples of invoking PL/SQL subprograms, see [Example 8-8](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/subprograms.htm#CIHHBJDE). For information about the CALL statement, see [*Oracle Database SQL Language Reference*](http://docs.oracle.com/cd/B28359_01/server.111/b28286/statements_4008.htm#SQLRF01108)

**Triggers**

A trigger is a stored subprogram associated with a table, view, or event. The trigger can be invoked once, when some event occurs, or many times, once for each row affected by an INSERT, UPDATE, or DELETE statement. The trigger can be invoked before or after the event.

The trigger in [Example 1-18](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/overview.htm#BABGHAIG) is invoked whenever salaries in the employees table are updated. For each update, the trigger writes a record to the emp\_audit table. ([Example 1-10](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/overview.htm#BABCBICG) would invoke this trigger.)

***Example 1-18 Creating a Trigger***

SQL> CREATE TABLE emp\_audit (

2 emp\_audit\_id NUMBER(6),

3 up\_date DATE,

4 new\_sal NUMBER(8,2),

5 old\_sal NUMBER(8,2)

6 );

Table created.

SQL>

SQL> **CREATE OR REPLACE TRIGGER audit\_sal**

2 **AFTER UPDATE OF salary**

3 **ON employees**

4 **FOR EACH ROW**

5 **BEGIN**

6 **INSERT INTO emp\_audit**

7 **VALUES(:old.employee\_id, SYSDATE, :new.salary, :old.salary);**

8 **END;**

9 /

Trigger created.

SQL>

For more information about triggers, see [Chapter 9, "Using Triggers."](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/triggers.htm#g1043102)

**PL/SQL Packages (APIs Written in PL/SQL)**

A PL/SQL package bundles logically related types, variables, cursors, and subprograms into a database object called a package. The package defines a simple, clear, interface to a set of related subprograms and types that can be accessed by SQL statements.

PL/SQL lets you access many predefined packages (see [Access to Predefined Packages](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/overview.htm#CJAJDHCF)) and to create your own packages.

A package usually has two parts: a specification and a body.

The specification defines the application programming interface (API); it declares the types, constants, variables, exceptions, cursors, and subprograms. To create a package specification, use the [CREATE PACKAGE Statement](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/create_package.htm#i2091914).

The body contains the SQL queries for cursors and the code for subprograms.To create a package body, use the [CREATE PACKAGE BODY Statement](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/create_package_body.htm#i2065383).

In [Example 1-19](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/overview.htm#BABEHEAB), the emp\_actions package contains two procedures that update the employees table and one function that provides information.

***Example 1-19 Creating a Package and Package Body***

SQL> **-- Package specification:**

SQL>

SQL> CREATE OR REPLACE PACKAGE emp\_actions AS

2

3 PROCEDURE hire\_employee (

4 employee\_id NUMBER,

5 last\_name VARCHAR2,

6 first\_name VARCHAR2,

7 email VARCHAR2,

8 phone\_number VARCHAR2,

9 hire\_date DATE,

10 job\_id VARCHAR2,

11 salary NUMBER,

12 commission\_pct NUMBER,

13 manager\_id NUMBER,

14 department\_id NUMBER

15 );

16

17 PROCEDURE fire\_employee (emp\_id NUMBER);

18

19 FUNCTION num\_above\_salary (emp\_id NUMBER) RETURN NUMBER;

20 END emp\_actions;

21 /

Package created.

SQL> **-- Package body:**

SQL>

SQL> CREATE OR REPLACE PACKAGE BODY emp\_actions AS

2

3 -- Code for procedure hire\_employee:

4

5 PROCEDURE hire\_employee (

6 employee\_id NUMBER,

7 last\_name VARCHAR2,

8 first\_name VARCHAR2,

9 email VARCHAR2,

10 phone\_number VARCHAR2,

11 hire\_date DATE,

12 job\_id VARCHAR2,

13 salary NUMBER,

14 commission\_pct NUMBER,

15 manager\_id NUMBER,

16 department\_id NUMBER

17 ) IS

18 BEGIN

19 INSERT INTO employees

20 VALUES (employee\_id,

21 last\_name,

22 first\_name,

23 email,

24 phone\_number,

25 hire\_date,

26 job\_id,

27 salary,

28 commission\_pct,

29 manager\_id,

30 department\_id);

31 END hire\_employee;

32

33 -- Code for procedure fire\_employee:

34

35 PROCEDURE fire\_employee (emp\_id NUMBER) IS

36 BEGIN

37 DELETE FROM employees

38 WHERE employee\_id = emp\_id;

39 END fire\_employee;

40

41 -- Code for function num\_above\_salary:

42

43 FUNCTION num\_above\_salary (emp\_id NUMBER) RETURN NUMBER IS

44 emp\_sal NUMBER(8,2);

45 num\_count NUMBER;

46 BEGIN

47 SELECT salary INTO emp\_sal

48 FROM employees

49 WHERE employee\_id = emp\_id;

50

51 SELECT COUNT(\*) INTO num\_count

52 FROM employees

53 WHERE salary > emp\_sal;

54

55 RETURN num\_count;

56 END num\_above\_salary;

57 END emp\_actions;

58 /

Package body created.

SQL>

To invoke a packaged subprogram, you must know only name of the package and the name and parameters of the subprogram (therefore, you can change the implementation details inside the package body without affecting the invoking applications).

[Example 1-20](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/overview.htm#BABGEGJG) invokes the emp\_actions package procedures hire\_employee and fire\_employee.

***Example 1-20 Invoking a Procedure in a Package***

SQL> CALL **emp\_actions.hire\_employee** (300, 'Belden', 'Enrique',

2 'EBELDEN', '555.111.2222',

3 '31-AUG-04', 'AC\_MGR', 9000,

4 .1, 101, 110);

Call completed.

SQL> BEGIN

2 DBMS\_OUTPUT.PUT\_LINE

3 ('Number of employees with higher salary: ' ||

4 TO\_CHAR(emp\_actions.num\_above\_salary(120)));

5

6 **emp\_actions.fire\_employee**(300);

7 END;

8 /

Number of employees with higher salary: 34

PL/SQL procedure successfully completed.

SQL>

Packages are stored in the database, where they can be shared by many applications. Invoking a packaged subprogram for the first time loads the whole package and caches it in memory, saving on disk I/O for subsequent invocations. Thus, packages enhance reuse and improve performance in a multiuser, multi-application environment.

For more information about packages, see [Chapter 10, "Using PL/SQL Packages."](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/packages.htm#CIHIJECJ)

**Conditional Compilation**

PL/SQL provides conditional compilation, which lets you customize the functionality in a PL/SQL application without having to remove any source code. For example, you can:

* Use the latest functionality with the latest database release and disable the new features to run the application against an older release of the database.
* Activate debugging or tracing functionality in the development environment and hide that functionality in the application while it runs at a production site.

For more information, see [Conditional Compilation](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/fundamentals.htm#BABIHIHF).

**Embedded SQL Statements**

Processing a SQL query with PL/SQL is like processing files with other languages. For example, a PERL program opens a file, reads the file contents, processes each line, then closes the file. In the same way, a PL/SQL program issues a query and processes the rows from the result set as shown in [Example 1-21](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/overview.htm#BABCFJII).

***Example 1-21 Processing Query Results in a LOOP***

SQL> BEGIN

2 FOR someone IN (SELECT \* FROM employees WHERE employee\_id < 120)

3 LOOP

4 DBMS\_OUTPUT.PUT\_LINE('First name = ' || someone.first\_name ||

5 ', Last name = ' || someone.last\_name);

6 END LOOP;

7 END;

8 /

First name = Steven, Last name = King

First name = Neena, Last name = Kochhar

First name = Lex, Last name = De Haan

First name = Alexander, Last name = Hunold

First name = Bruce, Last name = Ernst

First name = David, Last name = Austin

First name = Valli, Last name = Pataballa

First name = Diana, Last name = Lorentz

First name = Nancy, Last name = Greenberg

First name = Daniel, Last name = Faviet

First name = John, Last name = Chen

First name = Ismael, Last name = Sciarra

First name = Jose Manuel, Last name = Urman

First name = Luis, Last name = Popp

First name = Den, Last name = Raphaely

First name = Alexander, Last name = Khoo

First name = Shelli, Last name = Baida

First name = Sigal, Last name = Tobias

First name = Guy, Last name = Himuro

First name = Karen, Last name = Colmenares

PL/SQL procedure successfully completed.

SQL>

You can use a simple loop like the one shown here, or you can control the process precisely by using individual statements to perform the query, retrieve data, and finish processing.

**Architecture of PL/SQL**

Topics:

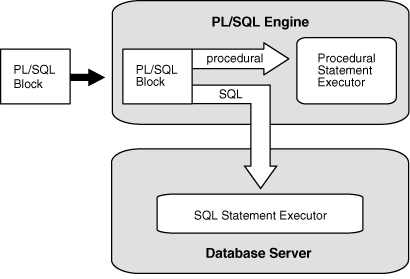
* [PL/SQL Engine](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/overview.htm#BABGEABB)
* [PL/SQL Units and Compilation Parameters](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/overview.htm#BABBDFJF)

**PL/SQL Engine**

The PL/SQL compilation and run-time system is an engine that compiles and executes PL/SQL units. The engine can be installed in the database or in an application development tool, such as Oracle Forms.

In either environment, the PL/SQL engine accepts as input any valid PL/SQL unit. The engine executes procedural statements, but sends SQL statements to the SQL engine in the database, as shown in [Figure 1-2](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/overview.htm#i6945).

***Figure 1-2 PL/SQL Engine***



Typically, the database processes PL/SQL units.

When an application development tool processes PL/SQL units, it passes them to its local PL/SQL engine. If a PL/SQL unit contains no SQL statements, the local engine processes the entire PL/SQL unit. This is useful if the application development tool can benefit from conditional and iterative control.

For example, Oracle Forms applications frequently use SQL statements to test the values of field entries and do simple computations. By using PL/SQL instead of SQL, these applications can avoid calls to the database.

**PL/SQL Units and Compilation Parameters**

A PL/SQL unit is any one of the following:

* PL/SQL block
* FUNCTION
* PACKAGE
* PACKAGE BODY
* PROCEDURE
* TRIGGER
* TYPE
* TYPE BODY

PL/SQL units are affected by PL/SQL compilation parameters (a category of database initialization parameters). Different PL/SQL units—for example, a package specification and its body—can have different compilation parameter settings.

[Table 1-1](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/overview.htm" \l "g3260498) lists and briefly describes the PL/SQL compilation parameters. For more information about these parameters, see [*Oracle Database Reference*](http://docs.oracle.com/cd/B28359_01/server.111/b28320/toc.htm).

To display the values of these parameters, use the static data dictionary view ALL\_PLSQL\_OBJECT\_SETTINGS. For more information about this view, see [*Oracle Database Reference*](http://docs.oracle.com/cd/B28359_01/server.111/b28320/statviews_2009.htm#REFRN20385).

***Table 1-1 PL/SQL Compilation Parameters***

| **Parameter** | **Description** |
| --- | --- |
| [PLSCOPE\_SETTINGS](http://docs.oracle.com/cd/B28359_01/server.111/b28320/initparams178.htm#REFRN10271)[Foot 1](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/overview.htm" \l "sthref89) | Controls the compile-time collection, cross reference, and storage of PL/SQL source code identifier data. Used by the PL/Scope tool, which is described in [*Oracle Database Advanced Application Developer's Guide*](http://docs.oracle.com/cd/B28359_01/appdev.111/b28424/adfns_plscope.htm#ADFNS022). |
| [PLSQL\_CCFLAGS](http://docs.oracle.com/cd/B28359_01/server.111/b28320/initparams179.htm#REFRN10261) 1 | Enables you to control conditional compilation of each PL/SQL unit independently. |
| [PLSQL\_CODE\_TYPE](http://docs.oracle.com/cd/B28359_01/server.111/b28320/initparams180.htm#REFRN10253) 1 | Specifies the compilation mode for PL/SQL units—INTERPRETED (the default) or NATIVE.  If the optimization level (set by PLSQL\_OPTIMIZE\_LEVEL) is less than 2:   * The compiler generates interpreted code, regardless ofPLSQL\_CODE\_TYPE. * If you specify NATIVE, the compiler warns you that NATIVEwas ignored. |
| [PLSQL\_DEBUG](http://docs.oracle.com/cd/B28359_01/server.111/b28320/initparams181.htm#REFRN10254) 1 | Specifies whether or not PL/SQL units will be compiled for debugging. See note following table. |
| [PLSQL\_NATIVE\_LIBRARY\_DIR](http://docs.oracle.com/cd/B28359_01/server.111/b28320/initparams182.htm#REFRN10168) | Has no effect. See note following table. |
| [PLSQL\_NATIVE\_LIBRARY\_SUBDIR\_COUNT](http://docs.oracle.com/cd/B28359_01/server.111/b28320/initparams183.htm#REFRN10169) | Has no effect. See note following table. |
| [PLSQL\_OPTIMIZE\_LEVEL](http://docs.oracle.com/cd/B28359_01/server.111/b28320/initparams184.htm#REFRN10255) 1 | Specifies the optimization level at which to compile PL/SQL units (the higher the level, the more optimizations the compiler tries to make).  If PLSQL\_OPTIMIZE\_LEVEL=1, PL/SQL units will be compiled for debugging. |
| [PLSQL\_WARNINGS](http://docs.oracle.com/cd/B28359_01/server.111/b28320/initparams186.htm#REFRN10249) 1 | Enables or disables the reporting of warning messages by the PL/SQL compiler, and specifies which warning messages to show as errors. |
| [NLS\_LENGTH\_SEMANTICS](http://docs.oracle.com/cd/B28359_01/server.111/b28320/initparams142.htm#REFRN10124) 1 | Enables you to create CHAR and VARCHAR2 columns using either byte or character length semantics. |

Footnote 1The compile-time value of this parameter is stored with the metadata of the PL/SQL unit.

**Note:**

The following compilation parameters are deprecated and might be unavailable in future Oracle Database releases:

* PLSQL\_DEBUG

For Release 11.1, it has the same effect as it had for Release 10.2—described in [Table 1-1](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/overview.htm#g3260498)—but the compiler warns you that it is deprecated.

Instead of PLSQL\_DEBUG, Oracle recommends using PLSQL\_OPTIMIZE\_LEVEL=1.

* PLSQL\_NATIVE\_LIBRARY\_DIR

For Release 11.1, it has no effect. The compiler does not warn you that it is deprecated.

* PLSQL\_NATIVE\_LIBRARY\_SUBDIR\_COUNT

For Release 11.1, it has no effect. The compiler does not warn you that it is deprecated.

The compile-time values of most of the parameters in [Table 1-1](http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/overview.htm#g3260498) are stored with the metadata of the PL/SQL unit, which means you can reuse those values when you explicitly recompile the program unit by doing the following:

1. Use one of the following statements to recompile the program unit:
   * ALTER FUNCTION COMPILE
   * ALTER PACKAGE COMPILE
   * ALTER PROCEDURE COMPILE
2. Include the REUSE SETTINGS clause in the statement.

This clause preserves the existing settings and uses them for the recompilation of any parameters for which values are not specified elsewhere in the statement.

If you use the SQL statement CREATE OR REPLACE to explicitly compile a PL/SQL subprogram, or if you do not include the REUSE SETTINGSclause in the ALTER COMPILE statement, then the value of the compilation parameter is its value for the session.