PART II
PL/SQL Program Structure

This part of the book presents the basic PL/SQL programming elements and statement constructs. Chapter 4 through Chapter 6 describe conditional (IF and CASE) and sequential control statements (e.g., GOTO and NULL), loops and the CONTINUE statement introduced for loops in Oracle Database 11*g*, and exception handling in the PL/SQL language. When you complete this section of the book, you will know how to construct blocks of code that correlate to the complex requirements in your applications.

CHAPTER 4

Conditional and Sequential Control

This chapter describes two types of PL/SQL control statements: conditional control statements and sequential control statements. Almost every piece of code you write will require conditional control, which is the ability to direct the flow of execution through your program based on a condition. You do this with IF-THEN-ELSE and CASE statements. There are also CASE expressions; while not the same as CASE statements, they can sometimes be used to eliminate the need for an IF or CASE statement altogether. Far less often, you will need to tell PL/SQL to transfer control unconditionally via the GOTO statement, or explicitly to do nothing via the NULL statement.

IF Statements

The IF statement allows you to implement conditional branching logic in your programs. With it, you'll be able to implement requirements such as:

- If the salary is between $10,000 and $20,000, apply a bonus of $1,500.
- If the collection contains more than 100 elements, truncate it.

The IF statement comes in three flavors, as shown in the following table.

IF type	Characteristics
`IF THEN END IF;`	This is the simplest form of the IF statement. The condition between IF and THEN determines whether the set of statements between THEN and END IF should be executed. If the condition evaluates to FALSE or NULL, the code is not executed.
`IF THEN ELSE END IF;`	This combination implements an either/or logic: based on the condition between the IF and THEN keywords, execute the code either between THEN and ELSE or between ELSE and END IF. One of these two sections of executable statements is performed.

IF type	Characteristics
`IF THEN EL SIF ELSE END IF;`	This last and most complex form of the IF statement selects a condition that is TRUE from a series of mutually exclusive conditions and then executes the set of statements associated with that condition. If you're writing IF statements like this using any release from Oracle9*i* Database Release 1 onward, you should consider using searched CASE statements instead.

The IF-THEN Combination

The general format of the IF-THEN syntax is as follows:

```
IF condition
THEN
   ... sequence of executable statements ...
END IF;
```

The *condition* is a Boolean variable, constant, or expression that evaluates to TRUE, FALSE, or NULL. If *condition* evaluates to TRUE, the executable statements found after the THEN keyword and before the matching END IF statement are executed. If *condition* evaluates to FALSE or NULL, those statements are not executed.

Three-Valued Logic

Boolean expressions can return three possible results. When all values in a Boolean expression are known, the result is either TRUE or FALSE. For example, there is no doubt when determining the truth or falsity of an expression such as:

```
(2 < 3) AND (5 < 10)
```

Sometimes, however, you don't know all values in an expression. That's because databases allow for values to be NULL, or missing. What, then, can be the result from an expression involving NULLs? For example:

```
2 < NULL
```

Because you don't know what the missing value is, the only answer you can give is "I don't know." This is the essence of so-called *three-valued logic*—you can have not only TRUE and FALSE as a possible result, but also NULL.

To learn more about three-valued logic, I recommend Lex de Haan and Jonathan Gennick's *Oracle Magazine* article "Nulls: Nothing to Worry About." You might find C. J. Date's book *Database in Depth: Relational Theory for the Practitioner* helpful as well. I'll also have more to say about three-valued logic as you go through this chapter.

The following IF condition compares two different numeric values. Remember that if one of these two values is NULL, then the entire expression returns NULL. In the following example, the bonus is not given when salary is NULL:

```
IF salary > 40000
THEN
   give_bonus (employee_id,500);
END IF;
```

There are exceptions to the rule that a NULL in a Boolean expression leads to a NULL result. Some operators and functions are specifically designed to deal with NULLs in a way that leads to TRUE and FALSE (and not NULL) results. For example, you can use IS NULL to test for the presence of a NULL:

```
IF salary > 40000 OR salary IS NULL
THEN
   give_bonus (employee_id,500);
END IF;
```

In this example, "salary IS NULL" evaluates to TRUE in the event that salary has no value, and otherwise to FALSE. Employees whose salaries are missing will now get bonuses too. (As indeed they probably should, considering their employer was so inconsiderate as to lose track of their pay in the first place.)

Using operators such as IS NULL and IS NOT NULL or functions such as COALESCE and NVL2 are good ways to detect and deal with potentially NULL values. For every variable that you reference in every Boolean expression that you write, be sure to think carefully about the consequences if that variable is NULL.

It's not necessary to put the IF, THEN, and END IF keywords on their own lines. In fact, line breaks don't matter at all for any type of IF statement. You could just as easily write:

```
IF salary > 40000 THEN give_bonus (employee_id,500); END IF;
```

Putting everything on one line is perfectly fine for simple IF statements such as the one shown here. However, when writing IF statements of any complexity at all, you'll find that readability is much greater when you format the statement such that each keyword begins a new line. For example, the following code would be very difficult to follow if it were all crammed on a single line. Actually, it's difficult to follow as it appears on three lines:

```
IF salary > 40000 THEN INSERT INTO employee_bonus (eb_employee_id, eb_bonus_amt)
VALUES (employee_id, 500); UPDATE emp_employee SET emp_bonus_given=1 WHERE emp_
employee_id=employee_id; END IF;
```

Ugh! Who'd want to spend time figuring that out? It's much more readable when formatted nicely:

```
IF salary > 40000
THEN
   INSERT INTO employee_bonus
      (eb_employee_id, eb_bonus_amt)
      VALUES (employee_id, 500);
```

```
   UPDATE emp_employee
   SET emp_bonus_given=1
   WHERE emp_employee_id=employee_id;
END IF;
```

This readability issue becomes even more important when using the ELSE and ELSIF keywords, and when nesting one IF statement inside another. Take full advantage of indents and formatting to make the logic of your IF statements easily decipherable. Future maintenance programmers will thank you.

The IF-THEN-ELSE Combination

Use the IF-THEN-ELSE format when you want to choose between two mutually exclusive actions. The format of this either/or version of the IF statement is as follows:

```
IF condition
THEN
   ... TRUE sequence of executable statements ...
ELSE
   ... FALSE/NULL sequence of executable statements ...
END IF;
```

The *condition* is a Boolean variable, constant, or expression. If *condition* evaluates to TRUE, the executable statements found after the THEN keyword and before the ELSE keyword are executed (the "TRUE sequence of executable statements"). If *condition* evaluates to FALSE or NULL, the executable statements that come after the ELSE keyword and before the matching END IF keywords are executed (the "FALSE/NULL sequence of executable statements").

The important thing to remember is that one of the two sequences of statements will *always* execute, because IF-THEN-ELSE is an either/or construct. Once the appropriate set of statements has been executed, control passes to the statement immediately following the END IF keyword.

Following is an example of the IF-THEN-ELSE construct that builds upon the IF-THEN example shown in the previous section:

```
IF salary <= 40000
THEN
   give_bonus (employee_id, 0);
ELSE
   give_bonus (employee_id, 500);
END IF;
```

In this example, employees with a salary greater than $40,000 will get a bonus of $500, while all other employees will get no bonus at all. Or will they? What happens if salary, for whatever reason, happens to be NULL for a given employee? In that case, the statements following the ELSE will be executed, and the employee in question will get the bonus that is supposed to go only to highly paid employees. That's not good (well, it was

good in the last section, but not now)! If the salary could be NULL, you can protect yourself against this problem using the NVL function:

```
IF NVL(salary,0) <= 40000
THEN
   give_bonus (employee_id, 0);
ELSE
   give_bonus (employee_id, 500);
END IF;
```

The NVL function will return zero any time salary is NULL, ensuring that any employees with a NULL salary also get a zero bonus (those poor employees).

Using Boolean Flags

Often, it's convenient to use Boolean variables as flags so that you don't need to evaluate the same Boolean expression more than once. When doing so, remember that the result of a Boolean expression can be assigned directly to a Boolean variable. For example, rather than writing:

```
IF :customer.order_total > max_allowable_order
THEN
   order_exceeds_balance := TRUE;
ELSE
   order_exceeds_balance := FALSE;
END IF;
```

you can instead (assuming neither variable could be NULL) write the following, much simpler expression:

```
order_exceeds_balance
   := :customer.order_total > max_allowable_order;
```

Now, whenever you need to test whether an order's total exceeds the maximum, you can write the following easily understandable IF statement:

```
IF order_exceeds_balance
THEN
...
```

If you have not had much experience with Boolean variables, it may take you a little while to learn how to integrate them smoothly into your code. It is worth the effort, though. The result is cleaner, more readable code.

The IF-THEN-ELSIF Combination

This last form of the IF statement comes in handy when you have to implement logic that has many alternatives; i.e., when it's not an either/or situation. The IF-ELSIF formulation provides a way to handle multiple conditions within a single IF statement. In

general, you should use ELSIF with mutually exclusive alternatives (i.e., when only one condition can be TRUE for any execution of the IF statement). The general format for this variation of IF is:

```
IF condition-1
THEN
   statements-1
ELSIF condition-N
THEN
   statements-N
[ELSE
   else_statements]
END IF;
```

Be very careful to use ELSIF, not ELSEIF. The inadvertent use of ELSEIF is a fairly common syntax error. ELSE IF (two words) doesn't work either.

Logically speaking, the IF-THEN-ELSIF construct is one way to implement CASE statement functionality in PL/SQL. Of course, if you are using Oracle9*i* Database onward, you are probably better off actually using a CASE statement (discussed later in this chapter).

Each ELSIF clause must have a THEN after its *condition*. Only the ELSE keyword does not need the THEN keyword. The ELSE clause in the IF-ELSIF is the "otherwise" of the statement. If none of the conditions evaluate to TRUE, the statements in the ELSE clause are executed. The ELSE clause is optional, though; you can code an IF-ELSIF that has only IF and ELSIF clauses. In such a case, if none of the conditions are TRUE, no statements inside the IF block are executed.

Following is an implementation of the complete bonus logic described at the beginning of this chapter using the IF-THEN-ELSIF combination:

```
IF salary BETWEEN 10000 AND 20000
THEN
   give_bonus(employee_id, 1500);
ELSIF salary BETWEEN 20000 AND 40000
THEN
   give_bonus(employee_id, 1000);
ELSIF salary > 40000
THEN
   give_bonus(employee_id, 500);
ELSE
   give_bonus(employee_id, 0);
END IF;
```

Avoiding IF Syntax Gotchas

Keep in mind these points about IF statement syntax:

Always match up an IF with an END IF
: In all three variations of the IF statement, you must close off the executable statements associated with the conditional structure with an END IF keyword.

You must have a space between the keywords END and IF
: If you type ENDIF instead of END IF, the compiler will get confused and give you the following hard-to-understand error message:

```
ORA-06550: line 14, column 4:

PLS-00103: Encountered the symbol ";" when expecting one of the following:
```

The ELSIF keyword should not have an embedded E
: If you type ELSEIF in place of ELSIF, the compiler will get confused and will not recognize the ELSEIF as part of the IF statement. Instead, the compiler will interpret ELSEIF as a variable or a procedure name.

Place a semicolon (;) only after the END IF keywords
: The keywords THEN, ELSE, and ELSIF should not have a semicolon after them. They are not standalone executable statements, and, unlike END IF, do not complete a statement. If you include a semicolon after these keywords, the compiler will issue messages indicating that it is looking for a statement of some kind before the semicolon.

The conditions in the IF-ELSIF are always evaluated in the order of first condition to last condition. If two conditions evaluate to TRUE, the statements for the first such condition are executed. With respect to the current example, a salary of $20,000 will result in a bonus of $1,500 even though that $20,000 salary also satisfies the condition for a $1,000 bonus (BETWEEN is inclusive). Once a condition evaluates to TRUE, the remaining conditions are not evaluated at all.

The CASE statement represents a better solution to the bonus problem than the IF-THEN-ELSIF solution shown in this section. See "CASE Statements and Expressions" on page 93.

Even though overlapping conditions are allowed in an IF-THEN-ELSIF statement, it's best to avoid them when possible. In my example, the original spec is a bit ambiguous about how to handle boundary cases such as $20,000. Assuming that the intent is to give the highest bonuses to the lowest-paid employees (which seems like a reasonable approach to me), I would dispense with the BETWEEN operator and use the following less-than/greater-than logic. Note that I've also dispensed with the ELSE clause just to illustrate that it is optional:

```
IF salary >= 10000 AND salary <= 20000
THEN
   give_bonus(employee_id, 1500);
ELSIF salary > 20000 AND salary <= 40000
THEN
   give_bonus(employee_id, 1000);
ELSIF salary > 40000
THEN
   give_bonus(employee_id, 400);
END IF;
```

By taking steps to avoid overlapping conditions in an IF-THEN-ELSIF, I am eliminating a possible (probable?) source of confusion for programmers who come after me. I also eliminate the possibility of inadvertent bugs being introduced as a result of someone's reordering the ELSIF clauses. Note, though, that if salary is NULL, then no code will be executed, because there is no ELSE section.

The language does not require that ELSIF conditions be mutually exclusive. Always be aware of the possibility that two or more conditions might apply to a given value, and that consequently the order of those ELSIF conditions might be important.

Nested IF Statements

You can nest any IF statement within any other IF statement. The following IF statement shows several layers of nesting:

```
IF condition1
THEN
   IF condition2
   THEN
      statements2
   ELSE
      IF condition3
      THEN
         statements3
      ELSIF condition4
      THEN
         statements4
      END IF;
   END IF;
END IF;
```

Nested IF statements are often necessary to implement complex logic rules, but you should use them carefully. Nested IF statements, like nested loops, can be very difficult to understand and debug. If you find that you need to nest more than three levels deep in your conditional logic, you should review that logic and see if there is a simpler way to code the same requirement. If not, then consider creating one or more local modules to hide the innermost IF statements.

A key advantage of the nested IF structure is that it defers evaluation of inner conditions. The conditions of an inner IF statement are evaluated only if the condition for the outer IF statement that encloses them evaluates to TRUE. Therefore, one obvious reason to nest IF statements is to evaluate one condition only when another condition is TRUE. For example, in my code to award bonuses, I might write the following:

```
IF award_bonus(employee_id) THEN
   IF print_check (employee_id) THEN
      DBMS_OUTPUT.PUT_LINE('Check issued for ' || employee_id);
   END IF;
END IF;
```

This is reasonable, because I want to print a message for each bonus check issued, but I don't want to print a bonus check for a zero amount in cases where no bonus was given.

Short-Circuit Evaluation

PL/SQL uses *short-circuit evaluation*, which means that PL/SQL need not evaluate all of the expression in an IF statement. For example, when evaluating the expression in the following IF statement, PL/SQL stops evaluation and immediately executes the ELSE branch if the first operand is either FALSE or NULL:

```
IF condition1 AND condition2
THEN
   ...
ELSE
   ...
END IF;
```

PL/SQL can stop evaluation of the expression when *condition1* is FALSE or NULL, because the THEN branch is executed only when the result of the expression is TRUE, and that requires *both* operands to be TRUE. As soon as one operand is found to be other than TRUE, there is no longer any chance for the THEN branch to be taken.

I found something interesting while researching PL/SQL's short-circuit behavior. The behavior that you get depends on the expression's context. Consider the following statement:

```
my_boolean := condition1 AND condition2
```

Unlike the case with an IF statement, when *condition1* is NULL, this expression will *not* short-circuit. Why not? Because the result could be either NULL or FALSE, depending on *condition2*. For an IF statement, NULL and FALSE both lead to the ELSE branch, so a short circuit can occur. But for an assignment, the ultimate value must be known, and short-circuiting in this case can (and will) occur only when *condition1* is FALSE.

Similar to the case with AND, if the first operand of an OR operation in an IF statement is TRUE, PL/SQL immediately executes the THEN branch:

```
IF condition1 OR condition2
THEN
   ...
ELSE
   ...
END IF;
```

This short-circuiting behavior can be useful when one of your conditions is particularly expensive in terms of CPU or memory utilization. In such a case, be sure to place that condition at the end of the set of conditions:

```
IF low_CPU_condition AND high_CPU_condition
THEN
   ...
END IF;
```

The *low_CPU_condition* is evaluated first, and if the result is enough to determine the end result of the AND operation (i.e., the result is FALSE), the more expensive condition will not be evaluated, and your application's performance is the better for that evaluation's not happening.

However, if you are *depending* on that second condition being evaluated, perhaps because you want the side effects from a stored function that the condition invokes, then you have a problem and you need to reconsider your design. I don't believe it's good to depend on side effects in this manner.

You can achieve the effect of short-circuit evaluation in a much more explicit manner using a nested IF statement:

```
IF low_CPU_condition
THEN
   IF high_CPU_condition
   THEN
      ...
   END IF;
END IF;
```

Now, *high_CPU_condition* is evaluated only if *low_CPU_condition* evaluates to TRUE. This is the same effect as short-circuit evaluation, but it's more obvious at a glance what's going on. It's also more obvious that my intent is to evaluate *low_CPU_condition* first.

Short-circuiting also applies to CASE statements and CASE expressions. These are described in the next section.

CASE Statements and Expressions

The CASE statement allows you to select one sequence of statements to execute out of many possible sequences. They have been part of the SQL standard since 1992, although Oracle SQL didn't support CASE until the release of Oracle8*i* Database, and PL/SQL didn't support CASE until Oracle9*i* Database Release 1. From this release onward, the following types of CASE statements are supported:

Simple CASE statement
: Associates each of one or more sequences of PL/SQL statements with a value. Chooses which sequence of statements to execute based on an expression that returns one of those values.

Searched CASE statement
: Chooses which of one or more sequences of PL/SQL statements to execute by evaluating a list of Boolean conditions. The sequence of statements associated with the first condition that evaluates to TRUE is executed.

NULL or UNKNOWN?

Earlier I stated that the result from a Boolean expression can be TRUE, FALSE, or NULL. In PL/SQL that is quite true, but in the larger realm of relational theory it's considered incorrect to speak of a NULL result from a Boolean expression. Relational theory says that a comparison to NULL, such as:

```
2 < NULL
```

yields the Boolean value UNKNOWN. And UNKNOWN is not the same as NULL. That PL/SQL refers to UNKNOWN as NULL is not something you should lose sleep over. I want you to be aware, though, that UNKNOWN is the *true* third value in three-valued logic. And now I hope you'll never be caught (as I have been a few times!) using the wrong term when discussing three-valued logic with experts on relational theory.

In addition to CASE statements, PL/SQL also supports CASE *expressions*. A CASE expression is very similar in form to a CASE statement and allows you to choose which of one or more expressions to evaluate. The result of a CASE expression is a single value, whereas the result of a CASE statement is the execution of a sequence of PL/SQL statements.

Simple CASE Statements

A simple CASE statement allows you to choose which of several sequences of PL/SQL statements to execute based on the results of a single expression. Simple CASE statements take the following form:

```
CASE expression
WHEN result1 THEN
   statements1
WHEN result2 THEN
   statements2
...
ELSE
   statements_else
END CASE;
```

The ELSE portion of the statement is optional. When evaluating such a CASE statement, PL/SQL first evaluates *expression*. It then compares the result of *expression* with *result1*. If the two results match, *statements1* is executed. Otherwise, *result2* is checked, and so forth.

Following is an example of a simple CASE statement that uses the employee type as a basis for selecting the proper bonus algorithm:

```
CASE employee_type
WHEN 'S' THEN
   award_salary_bonus(employee_id);
WHEN 'H' THEN
   award_hourly_bonus(employee_id);
WHEN 'C' THEN
   award_commissioned_bonus(employee_id);
ELSE
   RAISE invalid_employee_type;
END CASE;
```

This CASE statement has an explicit ELSE clause; however, the ELSE is optional. When you do not explicitly specify an ELSE clause of your own, PL/SQL implicitly uses the following:

```
ELSE
   RAISE CASE_NOT_FOUND;
```

In other words, if you don't specify an ELSE clause, and none of the results in the WHEN clauses match the result of the CASE expression, PL/SQL raises a CASE_NOT_FOUND error. This behavior is different from what I'm used to with IF statements. When an IF statement lacks an ELSE clause, nothing happens when the condition is not met. With CASE, the analogous situation leads to an error.

By now you're probably wondering how, or even whether, the bonus logic shown earlier in this chapter can be implemented using a simple CASE statement. At first glance, it doesn't appear possible. However, a bit of creative thought yields the following solution:

```
CASE TRUE
WHEN salary >= 10000 AND salary <=20000
THEN
   give_bonus(employee_id, 1500);
WHEN salary > 20000 AND salary <= 40000
```

```
THEN
   give_bonus(employee_id, 1000);
WHEN salary > 40000
THEN
   give_bonus(employee_id, 500);
ELSE
   give_bonus(employee_id, 0);
END CASE;
```

The key point to note here is that the *expression* and *result* elements shown in the earlier syntax diagram can be either scalar values or expressions that evaluate to scalar values.

If you look back to the earlier IF-THEN-ELSIF statement implementing this same bonus logic, you'll see that I specified an ELSE clause for the CASE implementation, whereas I didn't specify an ELSE for the IF-THEN-ELSIF solution. The reason for the addition of the ELSE is simple: if no bonus conditions are met, the IF statement does nothing, effectively resulting in a zero bonus. A CASE statement, however, will raise an error if no conditions are met—hence the need to code explicitly for the zero bonus case.

To avoid CASE_NOT_FOUND errors, be sure that it's impossible for one of your conditions not to be met.

While my previous CASE TRUE statement may look like a clever hack, it's really an explicit implementation of the searched CASE statement, which I talk about in the next section.

Searched CASE Statements

A searched CASE statement evaluates a list of Boolean expressions and, when it finds an expression that evaluates to TRUE, executes a sequence of statements associated with that expression. Essentially, a searched CASE statement is the equivalent of the CASE TRUE statement shown in the previous section.

Searched CASE statements have the following form:

```
CASE
WHEN expression1 THEN
   statements1
WHEN expression2 THEN
   statements2
...
ELSE
   statements_else
END CASE;
```

A searched CASE statement is a perfect fit for the problem of implementing the bonus logic. For example:

```
CASE
WHEN salary >= 10000 AND salary <=20000 THEN
   give_bonus(employee_id, 1500);
WHEN salary > 20000 AND salary <= 40000 THEN
   give_bonus(employee_id, 1000);
WHEN salary > 40000 THEN
   give_bonus(employee_id, 500);
ELSE
   give_bonus(employee_id, 0);
END CASE;
```

As with simple CASE statements, the following rules apply:

- Execution ends once a sequence of statements has been executed. If more than one expression evaluates to TRUE, only the statements associated with the first such expression are executed.
- The ELSE clause is optional. If no ELSE is specified, and no expressions evaluate to TRUE, then a CASE_NOT_FOUND exception is raised.
- WHEN clauses are evaluated in order, from top to bottom.

Following is an implementation of my bonus logic that takes advantage of the fact that WHEN clauses are evaluated in the order in which I write them. The individual expressions are simpler, but is the intent of the statement as easily grasped?

```
CASE
WHEN salary > 40000 THEN
   give_bonus(employee_id, 500);
WHEN salary > 20000 THEN
   give_bonus(employee_id, 1000);
WHEN salary >= 10000 THEN
   give_bonus(employee_id, 1500);
ELSE
   give_bonus(employee_id, 0);
END CASE;
```

If a given employee's salary is $20,000, then the first expression and second expression will evaluate to FALSE. The third expression will evaluate to TRUE, and that employee will be awarded a bonus of $1,500. If an employee's salary is $21,000, then the second expression will evaluate to TRUE, and the employee will be awarded a bonus of $1,000. Execution of the CASE statement will cease with the first WHEN condition that evaluates to TRUE, so a salary of $21,000 will never reach the third condition.

It's arguable whether you should take this approach to writing CASE statements. You should certainly be aware that it's possible to write such a statement, and you should

watch for such order-dependent logic in programs that you are called upon to modify or debug.

Order-dependent logic can be a subtle source of bugs when you decide to reorder the WHEN clauses in a CASE statement. Consider the following searched CASE statement in which, assuming a salary of $20,000, both WHEN expressions evaluate to TRUE:

```
CASE
WHEN salary BETWEEN 10000 AND 20000 THEN
   give_bonus(employee_id, 1500);
WHEN salary BETWEEN 20000 AND 40000 THEN
   give_bonus(employee_id, 1000);
...
```

Imagine the results if a future programmer unthinkingly decides to make the code neater by reordering the WHEN clauses in descending order by salary. Don't scoff at this possibility! We programmers frequently fiddle with perfectly fine, working code to satisfy some inner sense of order. Following is the CASE statement rewritten with the WHEN clauses in descending order:

```
CASE
WHEN salary BETWEEN 20000 AND 40000 THEN
   give_bonus(employee_id, 1000);
WHEN salary BETWEEN 10000 AND 20000 THEN
   give_bonus(employee_id, 1500);
...
```

Looks good, doesn't it? Unfortunately, because of the slight overlap between the two WHEN clauses, I've introduced a subtle bug into the code. Now an employee with a salary of $20,000 gets a bonus of $1,000 rather than the intended $1,500. There may be cases where overlap between WHEN clauses is desirable, but avoid it when feasible. Always remember that order matters, and resist the urge to fiddle with working code. *If it ain't broke, don't fix it.*

Since WHEN clauses are evaluated in order, you may be able to squeeze some extra efficiency out of your code by listing the most likely WHEN clauses first. In addition, if you have WHEN clauses with "expensive" expressions (e.g., requiring lots of CPU and memory), you may want to list those last in order to minimize the chances that they will be evaluated. See "Nested IF Statements" on page 90 for an example of this issue.

Use searched CASE statements when you want to use Boolean expressions as a basis for identifying a set of statements to execute. Use simple CASE statements when you can base that decision on the result of a single expression.

Nested CASE Statements

CASE statements can be nested just as IF statements can. For example, this rather difficult to follow implementation of my bonus logic uses a nested CASE statement:

```
CASE
WHEN salary >= 10000 THEN
   CASE
   WHEN salary <= 20000 THEN
      give_bonus(employee_id, 1500);
   WHEN salary > 40000 THEN
      give_bonus(employee_id, 500);
   WHEN salary > 20000 THEN
      give_bonus(employee_id, 1000);
   END CASE;
WHEN salary < 10000 THEN
   give_bonus(employee_id,0);
END CASE;
```

Any type of statement may be used within a CASE statement, so I could replace the inner CASE statement with an IF statement. Likewise, any type of statement, including CASE statements, may be nested within an IF statement.

CASE Expressions

CASE expressions do for expressions what CASE statements do for statements. Simple CASE expressions let you choose an expression to evaluate based on a scalar value that you provide as input. Searched CASE expressions evaluate a list of expressions to find the first one that evaluates to TRUE, and then return the results of an associated expression.

CASE expressions take the following two forms:

```
Simple_Case_Expression :=
   CASE expression
   WHEN result1 THEN
      result_expression1
   WHEN result2 THEN
      result_expression2
   ...
   ELSE
      result_expression_else
   END;
Searched_Case_Expression :=
   CASE
   WHEN expression1 THEN
      result_expression1
   WHEN expression2 THEN
      result_expression2
   ...
   ELSE
```

```
      result_expression_else
   END;
```

A CASE expression returns a single value, the result of whichever *result_expression* is chosen. Each WHEN clause must be associated with exactly one expression (no statements). Do not use semicolons or END CASE to mark the end of the CASE expression. CASE expressions are terminated by a simple END.

Following is an example of a simple CASE expression being used with the DBMS_OUTPUT package to output the value of a Boolean variable. (Recall that the PUT_LINE program is not overloaded to handle Boolean types.) In this example, the CASE expression converts the Boolean value into a character string, which PUT_LINE can then handle:

```
DECLARE
   boolean_true BOOLEAN := TRUE;
   boolean_false BOOLEAN := FALSE;
   boolean_null BOOLEAN;
   FUNCTION boolean_to_varchar2 (flag IN BOOLEAN) RETURN VARCHAR2 IS
   BEGIN
      RETURN
         CASE flag
            WHEN TRUE THEN 'True'
            WHEN FALSE THEN 'False'
            ELSE 'NULL'
         END;
   END;
BEGIN
   DBMS_OUTPUT.PUT_LINE(boolean_to_varchar2(boolean_true));
   DBMS_OUTPUT.PUT_LINE(boolean_to_varchar2(boolean_false));
   DBMS_OUTPUT.PUT_LINE(boolean_to_varchar2(boolean_null));
END;
```

A searched CASE expression can be used to implement my bonus logic, returning the proper bonus value for any given salary:

```
DECLARE
  salary NUMBER := 20000;
  employee_id NUMBER := 36325;
  PROCEDURE give_bonus (emp_id IN NUMBER, bonus_amt IN NUMBER) IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE(emp_id);
    DBMS_OUTPUT.PUT_LINE(bonus_amt);
  END;
BEGIN
   give_bonus(employee_id,
           CASE
              WHEN salary >= 10000 AND salary <= 20000 THEN 1500
              WHEN salary > 20000 AND salary <= 40000 THEN 1000
              WHEN salary > 40000 THEN 500
              ELSE 0
```

```
            END);
END;
```

You can use a CASE expression anywhere you can use any other type of expression or value. The following example uses a CASE expression to compute a bonus amount, multiplies that amount by 10, and assigns the result to a variable that is displayed via DBMS_OUTPUT:

```
DECLARE
  salary NUMBER := 20000;
  employee_id NUMBER := 36325;
  bonus_amount NUMBER;
BEGIN
   bonus_amount :=
      CASE
         WHEN salary >= 10000 AND salary <= 20000 THEN 1500
         WHEN salary > 20000 AND salary <= 40000 THEN 1000
         WHEN salary > 40000 THEN 500
         ELSE 0
      END * 10;
   DBMS_OUTPUT.PUT_LINE(bonus_amount);
END;
```

Unlike with the CASE statement, no error is raised in the event that no WHEN clause is selected in a CASE expression. Instead, when no WHEN conditions are met, a CASE expression will return NULL.

The GOTO Statement

The GOTO statement performs unconditional branching to another executable statement in the same execution section of a PL/SQL block. As with other constructs in the language, if you use GOTO appropriately and with care, your programs will be stronger for it.

The general format for a GOTO statement is:

```
GOTO label_name;
```

where *label_name* is the name of a label identifying the target statement. This GOTO label is defined in the program as follows:

```
<<label_name>>
```

You must surround the label name with double enclosing angle brackets (<< >>). When PL/SQL encounters a GOTO statement, it immediately shifts control to the first executable statement following the label. Following is a complete code block containing both a GOTO and a label:

```
BEGIN
   GOTO second_output;
   DBMS_OUTPUT.PUT_LINE('This line will never execute.');
```

```
   <<second_output>>
   DBMS_OUTPUT.PUT_LINE('We are here!');
END;
```

There are several restrictions on the GOTO statement:

- At least one executable statement must follow a label.
- The target label must be in the same scope as the GOTO statement.
- The target label must be in the same part of the PL/SQL block as the GOTO.

Contrary to popular opinion (including mine), the GOTO statement can come in handy. There are cases where a GOTO statement can simplify the logic in your program. On the other hand, because PL/SQL provides so many different control constructs and modularization techniques, you can almost always find a better way to do something than with a GOTO.

The NULL Statement

Usually when you write a statement in a program, you want it to do something. There are cases, however, when you want to tell PL/SQL to do absolutely nothing, and that is where the NULL statement comes in handy. The NULL statement has the following format:

```
NULL;
```

Well, you wouldn't want a do-nothing statement to be complicated, would you? The NULL statement is simply the reserved word NULL followed by a semicolon (;) to indicate that this is a statement and not a NULL value. The NULL statement does nothing except pass control to the next executable statement.

Why would you want to use the NULL statement? There are several reasons, described in the following sections.

Improving Program Readability

Sometimes, it's helpful to avoid any ambiguity inherent in an IF statement that doesn't cover all possible cases. For example, when you write an IF statement, you do not have to include an ELSE clause. To produce a report based on a selection, you can code:

```
IF :report_mgr.selection = 'DETAIL'
THEN
   exec_detail_report;
END IF;
```

What should the program be doing if the report selection is not 'DETAIL'? One might assume that the program is supposed to do nothing. But because this is not explicitly stated in the code, you are left to wonder if perhaps there was an oversight. If, on the

other hand, you include an explicit ELSE clause that does nothing, you state very clearly, "Don't worry, I thought about this possibility and I really want nothing to happen":

```
IF :report_mgr.selection = 'DETAIL'
THEN
   exec_detail_report;
ELSE
   NULL; -- Do nothing
END IF;
```

My example here was of an IF statement, but the same principle applies when you're writing CASE statements and CASE expressions. Similarly, if you want to temporarily remove all the code from a function or procedure, and yet still invoke that function or procedure, you can use NULL as a placeholder. Otherwise, you cannot compile a function or procedure without having any lines of code within it.

Using NULL After a Label

In some cases, you can pair NULL with GOTO to avoid having to execute additional statements. Most of you will never have to use the GOTO statement; there are very few occasions where it is truly needed. If you ever do use GOTO, however, you should remember that when you GOTO a label, at least one executable statement must follow that label. In the following example, I use a GOTO statement to quickly move to the end of my program if the state of my data indicates that no further processing is required:

```
PROCEDURE process_data (data_in IN orders%ROWTYPE,
                        data_action IN VARCHAR2)
IS
   status INTEGER;
BEGIN
   -- First in series of validations.
   IF data_in.ship_date IS NOT NULL
   THEN
      status := validate_shipdate (data_in.ship_date);
      IF status != 0 THEN GOTO end_of_procedure; END IF;
   END IF;

   -- Second in series of validations.
   IF data_in.order_date IS NOT NULL
   THEN
      status := validate_orderdate (data_in.order_date);
      IF status != 0 THEN GOTO end_of_procedure; END IF;
   END IF;

   ... more validations ...

   <<end_of_procedure>>
   NULL;
END;
```

With this approach, if I encounter an error in any single section, I use the GOTO to bypass all remaining validation checks. Because I do not have to do anything at the termination of the procedure, I place a NULL statement after the label because at least one executable statement is required there. Even though NULL does nothing, it is still an executable statement.