CHAPTER 11
Records

A *record* is a composite data structure, which means that it is composed of more than one element or component, each with its own value. Records in PL/SQL programs are very similar in concept and structure to the rows of a database table. The record as a whole does not have a value of its own; instead, each individual component or field has a value, and the record gives you a way to store and access these values as a group. Records can greatly simplify your life as a programmer, allowing you to write and manage your code more efficiently by shifting from field-level declarations and manipulation to record-level operations.

Records in PL/SQL

Each row in a table has one or more columns of various datatypes. Similarly, a record is composed of one or more fields. There are three different ways to define a record, but once defined, the same rules apply for referencing and changing fields in a record.

The following block demonstrates the declaration of a record that is based directly on an underlying database table. Suppose that I have defined a table to keep track of my favorite books:

```
CREATE TABLE books (
  book_id         INTEGER,
  isbn            VARCHAR2(13),
  title           VARCHAR2(200),
  summary         VARCHAR2(2000),
  author          VARCHAR2(200),
  date_published  DATE,
  page_count      NUMBER
);
```

I can then easily create a record based on this table, populate it with a query from the database, and then access the individual columns through the record's fields:

```
DECLARE
   my_book  books%ROWTYPE;
BEGIN
   SELECT *
     INTO my_book
     FROM books
    WHERE title = 'Oracle PL/SQL Programming, 6th Edition';

   IF my_book.author LIKE '%Feuerstein%'
   THEN
      DBMS_OUTPUT.put_line ('Our newest ISBN is ' || my_book.isbn);
   END IF;
END;
```

I can also define my own record type and use that as the basis for declaring records. Suppose, for example, that I want to work only with the author and title of a book. Rather than use %ROWTYPE to declare my record, I will instead create a record type:

```
DECLARE
   TYPE author_title_rt IS RECORD (
      author books.author%TYPE
     ,title books.title%TYPE
     );
   l_book_info author_title_rt;
BEGIN
   SELECT author, title INTO l_book_info
     FROM books WHERE isbn = '978-1-449-32445-2';
```

Let's take a look at some of the benefits of using records. Then I'll examine in more detail the different ways to define a record, and finish up with examples of using records in my programs.

Benefits of Using Records

The record data structure provides a high-level way of addressing and manipulating data defined inside PL/SQL programs (as opposed to stored in database tables). This approach offers several benefits, described in the following sections.

Data abstraction

When you abstract something, you generalize it, distancing yourself from the nitty-gritty details and concentrating on the big picture. When you create a module, you abstract the individual actions of the module into a name. The name (and program specification) represents those actions.

When you create a record, you abstract all the different attributes or fields of the subject of that record. You establish a relationship between those different attributes and give that relationship a name by defining a record.

Aggregate operations

Once you have stored information in records, you can perform operations on whole blocks of data at a time, rather than on each individual attribute. This kind of aggregate operation reinforces the abstraction of the record. Very often, you are not really interested in making changes to individual components of a record, but instead to the object that represents all of those different components.

Suppose that in my job I need to work with companies. I don't really care about whether a company has two lines of address information or three; instead, I want to work at the level of the company itself, making changes to, deleting, or analyzing its status. In all of these cases I am talking about a whole row in the database, not any specific column. The company record hides all that information from me, yet makes it accessible if and when I need it.

This orientation brings you closer to viewing your data as a collection of objects, with rules applied to those objects.

Leaner, cleaner code

Using records also helps you to write cleaner code, and less of it. When I use records, I invariably produce programs that have fewer lines of code, are less vulnerable to change, and need fewer comments. Records also cut down on variable sprawl; instead of declaring many individual variables, I declare a single record. This lack of clutter creates aesthetically attractive code that requires fewer resources to maintain.

Use of PL/SQL records can have a dramatic, positive impact on your programs, both in initial development and in ongoing maintenance. To ensure that I get the most out of record structures, I have set the following guidelines for my code development:

Create corresponding cursors and records
: Whenever I create a cursor in my programs, I also create a corresponding record (except in the case of cursor FOR loops). I always FETCH into a record, rather than into individual variables. In those few instances when this involves a little extra work, I marvel at the elegance of the approach and compliment myself on my commitment to principle. And starting with Oracle9*i* Database Release 2, I can even use records with DML statements!

Create table-based records
: Whenever I need to store table-based data within my programs, I create a new (or use a predefined) table-based record to store that data. That way, I only have to declare a single variable. Even better, the structure of that record will automatically adapt to changes in the table with each compilation.

Pass records as parameters
: Whenever appropriate, I pass records rather than individual variables as parameters in my procedural interfaces. This way, my procedure calls are less likely to change over time, making my code more stable.

Cursors are discussed in more detail in Chapter 15. They are, however, used so commonly with records that they appear in many of this chapter's examples.

Declaring Records

You can declare a record in one of three ways:

Table-based record
: Use the %ROWTYPE attribute with a table name to declare a record in which each field corresponds to—and has the same name as—a column in a table. In the following example, I declare a record named one_book with the same structure as the books table:

```
DECLARE
   one_book books%ROWTYPE;
```

Cursor-based record
: Use the %ROWTYPE attribute with an explicit cursor or cursor variable in which each field corresponds to a column or aliased expression in the cursor SELECT statement. In the following example, I declare a record with the same structure as an explicit cursor:

```
DECLARE
   CURSOR my_books_cur IS
      SELECT * FROM books
       WHERE author LIKE '%FEUERSTEIN%';

   one_SF_book my_books_cur%ROWTYPE;
```

Programmer-defined record
: Use the TYPE...RECORD statement to define a record in which each field is defined explicitly (with its name and datatype) in the TYPE statement for that record; a field in a programmer-defined record can even be another record. In the following example, I declare a record TYPE containing some information about my book-writing career and an "instance" of that type, a record:

```
DECLARE
   TYPE book_info_rt IS RECORD (
      author books.author%TYPE,
      category VARCHAR2(100),
      total_page_count POSITIVE);

   steven_as_author book_info_rt;
```

Notice that when I declare a record based on a record TYPE, I do not use the %ROWTYPE attribute. The book_info_rt element already is a TYPE.

The general format of the %ROWTYPE declaration is:

```
record_name [schema_name.]object_name%ROWTYPE
   [ DEFAULT|:= compatible_record ];
```

The *schema_name* is optional (if not specified, then the schema under which the code is compiled is used to resolve the reference). The *object_name* can be an explicit cursor, cursor variable, table, view, or synonym. You can provide an optional default value, which would be a record of the same or a compatible type.

Here is an example of the creation of a record based on a cursor variable:

```
DECLARE
   TYPE book_rc IS REF CURSOR RETURN books%ROWTYPE;
   book_cv book_rc;

   one_book book_cv%ROWTYPE;
BEGIN
   ...
```

The other way to declare and use a record is to do so implicitly, with a cursor FOR loop. In the following block, the book_rec record is not defined in the declaration section; PL/SQL automatically declares it for me with the %ROWTYPE attribute against the loop's query:

```
BEGIN
   FOR book_rec IN (SELECT * FROM books)
   LOOP
      calculate_total_sales (book_rec);
   END LOOP;
END;
```

By far the most interesting and complicated way to declare a record is with the TYPE statement, so let's explore that feature in a bit more detail.

Programmer-Defined Records

Table- and cursor-based records are great when you need to create program data matching those structures. Yet do these kinds of records cover all of our needs for composite data structures? What if I want to create a record that has nothing to do with either a table or a cursor? What if I want to create a record whose structure is derived from several different tables and views? Should I really have to create a "dummy" cursor just so I can end up with a record of the desired structure? For just these kinds of situations, PL/SQL offers programmer-defined records, declared with the TYPE...RECORD statement.

With the programmer-defined record, you have complete control over the number, names, and datatypes of fields in the record. To declare a programmer-defined record, you must perform two distinct steps:

1. Declare or define a record TYPE containing the structure you want in your record.
2. Use this record TYPE as the basis for declarations of your own actual records having that structure.

Declaring programmer-defined record TYPEs

You declare a record type with the TYPE statement. The TYPE statement specifies the name of the new record structure, and the components or fields that make up that record. The general syntax of the record TYPE definition is:

```
TYPE type_name IS RECORD
   (field_name1 datatype1 [[NOT NULL]:=|DEFAULT default_value],
    field_name2 datatype2 [[NOT NULL]:=|DEFAULT default_value],
    ...
    field_nameN datatypeN [[NOT NULL]:=|DEFAULT default_value]
   );
```

where *field_nameN* is the name of the *N*th field in the record, and *datatypeN* is the datatype of that *N*th field. The datatype of a record's field can be any of the following:

- A hardcoded scalar datatype (VARCHAR2, NUMBER, etc.).
- A programmer-defined SUBTYPE.
- An anchored declaration using %TYPE or %ROWTYPE attributes. In the latter case, I have created a *nested record*—one record inside another.
- A PL/SQL collection type; a field in a record can be a list or even a collection.
- A REF CURSOR, in which case the field contains a cursor variable.

Here is an example of a record TYPE statement:

```
TYPE company_rectype IS RECORD (
    comp# company.company_id%TYPE
  , list_of_names DBMS_SQL.VARCHAR2S
  , dataset SYS_REFCURSOR
  );
```

You can declare a record TYPE in a local declaration section or in a package specification; the latter approach allows you to globally reference that record type in any PL/SQL block compiled in the schema that owns the package or in the PL/SQL blocks of any schema that has EXECUTE privileges on the package.

Declaring the record

Once you have created your own customized record types, you can use those types in declarations of specific records. The actual record declarations have the following format:

```
record_name record_type;
```

where *record_name* is the name of the record, and *record_type* is the name of a record type that you have defined with the TYPE...RECORD statement.

To build a customer sales record, for example, I first define a record type called customer_sales_rectype, as follows:

```
PACKAGE customer_sales_pkg
IS
    TYPE customer_sales_rectype IS RECORD
       (customer_id   customer.customer_id%TYPE,
        customer_name customer.name%TYPE,
        total_sales   NUMBER (15,2)
       );
```

This is a three-field record structure that contains the primary key and name information for a customer, as well as a calculated, total amount of sales for the customer. I can then use this new record type to declare records with the same structure as this type:

```
DECLARE
   prev_customer_sales_rec customer_sales_pkg.customer_sales_rectype;
   top_customer_rec customer_sales_pkg.customer_sales_rectype;
```

Notice that I do not need the %ROWTYPE attribute, or any other kind of keyword, to denote this as a record declaration. The %ROWTYPE attribute is needed only for table and cursor records.

You can also pass records based on these types as arguments to procedures; simply use the record type as the type of the formal parameter, as shown here:

```
PROCEDURE analyze_cust_sales (
   sales_rec_in IN customer_sales_pkg.customer_sales_rectype)
```

In addition to specifying the datatype, you can supply default values for individual fields in a record with the DEFAULT or := syntax. Finally, each field name within a record must be unique.

Examples of programmer-defined record declarations

Suppose that I declare the following subtype, a cursor, and an associative array data structure:[1]

1. *Associative array* is the latest name for what used to be called a “PL/SQL table” or an “index-by table,” as explained in detail in Chapter 12.

```
SUBTYPE long_line_type IS VARCHAR2(2000);

CURSOR company_sales_cur IS
   SELECT name, SUM (order_amount) total_sales
     FROM company c, orders o
    WHERE c.company_id = o.company_id;

TYPE employee_ids_tabletype IS
   TABLE OF employees.employee_id%TYPE
   INDEX BY BINARY_INTEGER;
```

I can then define the following types of programmer-defined record in that same declaration section:

- A programmer-defined record that is a subset of the company table, plus a PL/SQL table of employees. I use the %TYPE attribute to link the fields in the record directly to the table. I then add a third field, which is actually an associative array of employee ID numbers:

  ```
  TYPE company_rectype IS RECORD
     (company_id    company.company_id%TYPE,
      company_name  company.name%TYPE,
      new_hires_tab employee_ids_tabletype);
  ```

- A mish-mash of a record that demonstrates the different kinds of field declarations in a record, including the NOT NULL constraint, the use of a subtype, the %TYPE attribute, a default value specification, an associative array, and a nested record. These varieties are shown here:

  ```
  TYPE mishmash_rectype IS RECORD
     (emp_number NUMBER(10) NOT NULL := 0,
      paragraph_text long_line_type,
      company_nm company.name%TYPE,
      total_sales company_sales.total_sales%TYPE := 0,
      new_hires_tab employee_ids_tabletype,
      prefers_nonsmoking_fl BOOLEAN := FALSE,
      new_company_rec company_rectype
     );
  ```

As you can see, PL/SQL offers tremendous flexibility in designing your own record structures. Your records can represent tables, views, and SELECT statements in a PL/SQL program. They can also be arbitrarily complex, with fields that are actually records within records or associative arrays.

Working with Records

Regardless of how you define a record (based on a table, cursor, or explicit record TYPE statement), you work with the resulting record in the same ways. You can work with the data in a record at the "record level," or you can work with individual fields of the record.

Record-level operations

When you work at the record level, you avoid any references to individual fields in the record. Here are the record-level operations currently supported by PL/SQL:

- You can copy the contents of one record to another, as long as they are defined based on the same user-defined record types or compatible %ROWTYPE records (they have the same number of fields and the same or implicitly convertible datatypes).
- You can assign a value of NULL to a record with a simple assignment.
- You can define and pass the record as an argument in a parameter list.
- You can RETURN a record back through the interface of a function.

Several record-level operations are not yet supported:

- You cannot use the IS NULL syntax to see if all fields in the record have NULL values. Instead, you must apply the IS NULL operator to each field individually.
- You cannot compare two records—for example, you cannot ask if the records (i.e., the values of their fields) are the same or different, or if one record is greater than or less than another. Unfortunately, to answer these kinds of questions, you must compare each field individually. I cover this topic and provide a utility that generates such comparison code in "Comparing Records" on page 337.
- Prior to Oracle9*i* Database Release 2, you could not insert into a database table with a record. Instead, you had to pass each individual field of the record for the appropriate column. For more information on record-based DML, see Chapter 14.

You can perform record-level operations on any records with compatible structures. In other words, the records must have the same number of fields and the same or convertible datatypes, but they don't have to be the same type. Suppose that I have created the following table:

```
CREATE TABLE cust_sales_roundup (
   customer_id NUMBER (5),
   customer_name VARCHAR2 (100),
   total_sales NUMBER (15,2)
   )
```

Then the three records defined as follows all have compatible structures, and I can "mix and match" the data in these records as shown:

```
DECLARE
   cust_sales_roundup_rec cust_sales_roundup%ROWTYPE;

   CURSOR cust_sales_cur IS SELECT * FROM cust_sales_roundup;
   cust_sales_rec cust_sales_cur%ROWTYPE;
   TYPE customer_sales_rectype IS RECORD
      (customer_id NUMBER(5),
       customer_name customer.name%TYPE,
       total_sales NUMBER(15,2)
      );
   preferred_cust_rec customer_sales_rectype;
BEGIN
   -- Assign one record to another.
   cust_sales_roundup_rec := cust_sales_rec;
   preferred_cust_rec := cust_sales_rec;
END;
```

Let's look at some other examples of record-level operations:

- In this example, I'll assign a default value to a record. You can initialize a record at the time of declaration by assigning it another compatible record. In the following program, I assign an IN argument record to a local variable. I might do this so that I can modify the values of fields in the record:

```
PROCEDURE compare_companies
   (prev_company_rec IN company%ROWTYPE)
IS
   curr_company_rec company%ROWTYPE := prev_company_rec;
BEGIN
   ...
END;
```

- In this next initialization example, I create a new record type and record. I then create a second record type using the first record type as its single column. Finally, I initialize this new record with the previously defined record:

```
DECLARE
   TYPE first_rectype IS RECORD (var1 VARCHAR2(100) := 'WHY NOT');
   first_rec first_rectype;
   TYPE second_rectype IS RECORD (nested_rec first_rectype := first_rec);
BEGIN
   ...
END;
```

- I can also perform assignments within the execution section, as you might expect. In the following example I declare two different rain_forest_history records and then set the current history information to the previous history record:

```
DECLARE
   prev_rain_forest_rec rain_forest_history%ROWTYPE;
   curr_rain_forest_rec rain_forest_history%ROWTYPE;
BEGIN
   ... initialize previous year rain forest data ...

   -- Transfer data from previous to current records.
   curr_rain_forest_rec := prev_rain_forest_rec;
```

- The result of this aggregate assignment is that the value of each field in the current record is set to the value of the corresponding field in the previous record. I could also have accomplished this with individual direct assignments from the previous to current records. This would have required multiple distinct assignments and lots of typing; whenever possible, use record-level operations to save time and make your code less vulnerable to change.
- I can move data directly from a row in a table to a record in a program by fetching directly into a record. Here are two examples:

  ```
  DECLARE
     /*
     || Declare a cursor and then define a record based on that cursor
     || with the %ROWTYPE attribute.
     */
     CURSOR cust_sales_cur IS
        SELECT customer_id, customer_name, SUM (total_sales) tot_sales
          FROM cust_sales_roundup
         WHERE sold_on < ADD_MONTHS (SYSDATE, -3)
        GROUP BY customer_id, customer_name;
     cust_sales_rec cust_sales_cur%ROWTYPE;
  BEGIN
     /* Move values directly into record by fetching from cursor */

     OPEN cust_sales_cur;
     FETCH cust_sales_cur INTO cust_sales_rec;
     CLOSE cust_sales_cur;
  ```

  In this next block, I declare a programmer-defined TYPE that matches the data retrieved by the implicit cursor. Then I SELECT directly into a record based on that type:

  ```
  DECLARE
     TYPE customer_sales_rectype IS RECORD
        (customer_id   customer.customer_id%TYPE,
         customer_name customer.name%TYPE,
         total_sales   NUMBER (15,2)
         );
     top_customer_rec  customer_sales_rectype;
  BEGIN
     /* Move values directly into the record: */
     SELECT customer_id, customer_name, SUM (total_sales)
       INTO top_customer_rec
       FROM cust_sales_roundup
      WHERE sold_on < ADD_MONTHS (SYSDATE, -3)
       GROUP BY customer_id, customer_name;
  ```

- I can set all fields of a record to NULL with a direct assignment:

  ```
  /* File on web: record_assign_null.sql */
  FUNCTION dept_for_name (
     department_name_in IN departments.department_name%TYPE
  )
  ```

```
      RETURN departments%ROWTYPE
   IS
      l_return   departments%ROWTYPE;

      FUNCTION is_secret_department (
         department_name_in IN departments.department_name%TYPE
      )
         RETURN BOOLEAN
      IS
      BEGIN
         RETURN CASE department_name_in
                    WHEN 'VICE PRESIDENT' THEN TRUE
                    ELSE FALSE
                 END;
      END is_secret_department;
   BEGIN
      SELECT *
        INTO l_return
        FROM departments
       WHERE department_name = department_name_in;

      IF is_secret_department (department_name_in)
      THEN
         l_return := NULL;
      END IF;

      RETURN l_return;
   END dept_for_name;
```

Whenever possible, try to work with records at the aggregate level—the record as a whole, not individual fields. The resulting code is much easier to write and maintain. There are, of course, many situations in which you need to manipulate individual fields of a record, though. Let's take a look at how you would do that.

Field-level operations

When you need to access a field within a record (to either read or change its value), you must use *dot notation*, just as you would when identifying a column from a specific database table. The syntax for such a reference is:

```
[[schema_name.]package_name.]record_name.field_name
```

You need to provide a package name only if the record is defined in the specification of a package that is different from the one you are working on at that moment. You need to provide a schema name only if the package is owned by a schema different from that in which you are compiling your code.

Once you have used dot notation to identify a particular field, all the normal rules in PL/SQL apply as to how you can reference and change the value of that field. Let's take a look at some examples.

The assignment operator (:=) changes the value of a particular field. In the first assignment, total_sales is zeroed out. In the second assignment, a function is called to return a value for the Boolean flag output_generated (it is set to TRUE, FALSE, or NULL):

```
BEGIN
   top_customer_rec.total_sales := 0;
   report_rec.output_generated := check_report_status (report_rec.report_id);
END;
```

In the next example I create a record based on the rain_forest_history table, populate it with values, and then insert a record into that same table:

```
DECLARE
   rain_forest_rec rain_forest_history%ROWTYPE;
BEGIN
   /* Set values for the record */
   rain_forest_rec.country_code  := 1005;
   rain_forest_rec.analysis_date := ADD_MONTHS (TRUNC (SYSDATE), -3);
   rain_forest_rec.size_in_acres := 32;
   rain_forest_rec.species_lost  := 425;

   /* Insert a row in the table using the record values */
   INSERT INTO rain_forest_history
          (country_code, analysis_date, size_in_acres, species_lost)
   VALUES
      (rain_forest_rec.country_code,
       rain_forest_rec.analysis_date,
       rain_forest_rec.size_in_acres,
       rain_forest_rec.species_lost);
   ...
END;
```

Notice that because the analysis_date field is of type DATE, I can assign any valid DATE expression to that field. The same goes for the other fields, and this is even true for more complex structures.

Starting with Oracle9*i* Database Release 2, you can also perform a record-level insert, simplifying the preceding INSERT statement into nothing more than this:

```
INSERT INTO rain_forest_history
   VALUES rain_forest_rec;
```

Record-level DML (for both inserts and updates) is covered fully in Chapter 14.

Field-level operations with nested records

Suppose that I have created a nested record structure; that is, one of the fields in my "outer" record is actually another record. In the following example I declare a record TYPE for all the elements of a telephone number (phone_rectype), and then declare a record TYPE that collects all the phone numbers for a person together in a single structure (contact_set_rectype):

```
DECLARE
   TYPE phone_rectype IS RECORD
      (intl_prefix   VARCHAR2(2),
       area_code     VARCHAR2(3),
       exchange      VARCHAR2(3),
       phn_number    VARCHAR2(4),
       extension     VARCHAR2(4)
      );

   -- Each field is a nested record...
  TYPE contact_set_rectype IS RECORD
      (day_phone#    phone_rectype,
       eve_phone#    phone_rectype,
       fax_phone#    phone_rectype,
       home_phone#   phone_rectype,
       cell_phone#   phone_rectype
      );

   auth_rep_info_rec contact_set_rectype;
BEGIN
```

Although I still use the dot notation to refer to a field with nested records, now I might have to refer to a field that is nested several layers deep inside the structure. To do this I must include an extra dot for each nested record structure, as shown in the following assignment, which sets the fax phone number's area code to the home phone number's area code:

```
auth_rep_info_rec.fax_phone#.area_code :=
   auth_rep_info_rec.home_phone#.area_code;
```

Field-level operations with package-based records

Finally, here is an example demonstrating references to packaged records (and package-based record TYPEs). Suppose that I want to plan out my summer reading (for all those days I will be lounging about in the sand outside my Caribbean hideaway). I create a package specification as follows:

```
CREATE OR REPLACE PACKAGE summer
IS
   TYPE reading_list_rt IS RECORD (
      favorite_author  VARCHAR2 (100),
      title            VARCHAR2 (100),
      finish_by        DATE);

   must_read reading_list_rt;
   wifes_favorite reading_list_rt;
END summer;

CREATE OR REPLACE PACKAGE BODY summer
IS
BEGIN  -- Initialization section of package
   must_read.favorite_author := 'Tepper, Sheri S.';
```

```
      must_read.title := 'Gate to Women''s Country';
   END summer;
```

With this package compiled in the database, I can then construct my reading list as follows:

```
   DECLARE
      first_book summer.reading_list_rt;
      second_book summer.reading_list_rt;
   BEGIN
      summer.must_read.finish_by := TO_DATE ('01-AUG-2009', 'DD-MON-YYYY');
      first_book := summer.must_read;

      second_book.favorite_author := 'Hobb, Robin';
      second_book.title := 'Assassin''s Apprentice';
      second_book.finish_by := TO_DATE ('01-SEP-2009', 'DD-MON-YYYY');
   END;
```

I declare two local book records. I then assign a "finish by" date to the packaged must-read book (notice the *package.record.field* syntax) and assign that packaged record to my first book of the summer record. I then assign values to individual fields for the second book of the summer.

Note that when you work with the UTL_FILE built-in package for file I/O in PL/SQL, you follow these same rules. The UTL_FILE.FILE_TYPE datatype is actually a record TYPE definition. So when you declare a file handle, you are really declaring a record of a package-based TYPE:

```
   DECLARE
      my_file_id UTL_FILE.FILE_TYPE;
```

Comparing Records

How can you check to see if two records are equal (i.e., that each corresponding field contains the same value)? It would be wonderful if PL/SQL would allow you to perform a direct comparison, as in:

```
   DECLARE
      first_book summer.reading_list_rt := summer.must_read;
      second_book summer.reading_list_rt := summer.wifes_favorite;
   BEGIN
      IF first_book = second_book /* THIS IS NOT SUPPORTED! */
      THEN
         lots_to_talk_about;
      END IF;
   END;
```

Unfortunately, you cannot do that. Instead, to test for record equality, you must write code that compares each field individually. If a record doesn't have many fields, this isn't too cumbersome. For the reading list record, you would write something like this:

```
DECLARE
   first_book summer.reading_list_rt := summer.must_read;
   second_book summer.reading_list_rt := summer.wifes_favorite;
BEGIN
   IF  first_book.favorite_author = second_book.favorite_author
      AND first_book.title = second_book.title
      AND first_book.finish_by = second_book.finish_by
   THEN
      lots_to_talk_about;
   END IF;
END;
```

There is one complication to keep in mind. If your requirements indicate that two NULL records are equal (equally NULL), you will have to modify each comparison to something like this:

```
(first_book.favorite_author = second_book.favorite_author
   OR( first_book.favorite_author IS NULL AND
       second_book.favorite_author IS NULL))
```

Any way you look at it, this is pretty tedious coding. Wouldn't it be great if you could generate code to do this for you? In fact, it's not all that difficult to do precisely that—at least if the records you want to compare are defined with %ROWTYPE against a table or view. In this case, you can obtain the names of all fields from the ALL_TAB_COLUMNS data dictionary view and then format the appropriate code out to the screen or to a file.

Better yet, you don't have to figure all that out yourself. Instead, you can download and run the "records equal" generator designed by Dan Spencer; you will find his package on the book's website in the *gen_record_comparison.pkg* file.

Trigger Pseudorecords

When you are writing code inside database triggers for a particular table, the database makes available to you two structures, OLD and NEW, which are *pseudorecords*. These structures have the same format as table-based records declared with %ROWTYPE—a field for every column in the table:

OLD
: This pseudorecord shows the values of each column in the table *before* the current transaction started.

NEW
: This pseudorecord reveals the new values of each column about to be placed in the table when the current transaction completes.

When you reference OLD and NEW within the body of the trigger, you must preface those identifiers with a colon; within the WHEN clause, however, do not use the colon. Here is an example:

```
TRIGGER check_raise
   AFTER UPDATE OF salary
   ON employee
   FOR EACH ROW
WHEN  (OLD.salary != NEW.salary) OR
      (OLD.salary IS NULL AND NEW.salary IS NOT NULL) OR
      (OLD.salary IS NOT NULL AND NEW.salary IS NULL)
BEGIN
   IF :NEW.salary > 100000 THEN ...
```

Chapter 19 offers a more complete explanation of how you can put the OLD and NEW pseudorecords to use in your database triggers. In particular, that chapter describes the many restrictions on how you can work with OLD and NEW.

%ROWTYPE and invisible columns (Oracle Database 12c)

As of 12.1, you can now define *invisible columns* in relational tables. An invisible column is a user-defined hidden column, which means that if you want to display or assign a value to an invisible column, you must specify its name explicitly. Here is an example of defining an invisible column in a table:

```
CREATE TABLE my_table (i INTEGER, d DATE, t TIMESTAMP INVISIBLE)
```

You can make an invisible column *visible* with an ALTER TABLE statement, as in:

```
ALTER TABLE my_table MODIFY t VISIBLE
```

The SELECT * syntax will not display an INVISIBLE column. However, if you include an INVISIBLE column in the select list of a SELECT statement, then the column will be displayed. You cannot implicitly specify a value for an INVISIBLE column in the VALUES clause of an INSERT statement. You must specify the INVISIBLE column in the column list. You must explicitly specify an INVISIBLE column in %ROWTYPE attributes.

The following block, for example, will fail to compile with the *PLS-00302: component 'T' must be declared* error:

```
DECLARE
    /* Record has two fields: i and d */
   l_data my_table%ROWTYPE;
BEGIN
   SELECT * INTO l_data FROM my_table;
   DBMS_OUTPUT.PUT_LINE ('t = ' || l_data.t);
END;
/
```

The problem is that since *T* is invisible, the record declared with %ROWTYPE contains only two fields, named *I* and *D*. I cannot reference a field named *T*.

If, however, I make that column visible, Oracle will then create a field for it in a %ROWTYPE-declared record. This also means that after you make an invisible column visible, Oracle will change the status of all program units that declare records using %ROWTYPE against that column's table to INVALID.