The discussion on PL/SQL was left at the end of Chapter 10, having introduced the block structure of the language and the various forms this structure may take. The simplest form is the anonymous block that has no name and is not stored on the server, although, for convenience, the client may store it in an external file.

Stored procedures and stored functions do have names and are stored as data items in a schema of the database. A package is also named and retained within the database. It generally consists of several procedures, functions, declared variables and cursors that share a related purpose and that may be referenced from within the package according to the scoping rules.

Triggers are always stored; because their activation relies on database events, it is essential that they are permanently available whenever the event occurs.

This chapter completes the survey of PL/SQL programming structures and techniques, illustrating more complex use of the language elements and introducing further constructs which give the seasoned programmer full control over the database operation.

All of the PL/SQL introduced here and in the previous discussion may be written, tested and stored in the database to create a library of standard routines. These are accessible by the front-end applications developer and offer an economic, reliable and powerful bank of reusable code.

13.1 MUTATING TABLES

One particular situation in which multiple trigger types are essential is in avoiding the so-called mutating table problem. This problem generally occurs when a row-
level trigger attempts to reference a data item in the trigger’s related SQL table through a \textit{select} statement rather than from the before-image or after-image of the current row.

A row-trigger is created to assure a complex legal contract drawn up between a club and its leading goal scorer. It ensures that he continues to be the league’s highest paid player whenever the salary of any other league player is increased, or any new player is registered in the league. This causes a \textit{mutating table problem} to occur. The update or insertion of other rows in the \textit{contracts} table fire the trigger but the trigger seeks to update the leading goal scorer’s contract salary not the player’s contract responsible for the event. An example of such a trigger is given in Figure 13.1. Its associated output, following an appropriate update request, is demonstrated in Figure 13.2.

```plsql
CREATE OR REPLACE TRIGGER contracts_biu_trigger
BEFORE INSERT OR UPDATE OF Salary ON Contracts
FOR EACH ROW

DECLARE
  v_max_sal  NUMBER;
  v_ufa_id   CHAR(8);
BEGIN
  SELECT ufa_id, salary INTO v_ufa_id, v_max_sal
  FROM Contracts
  WHERE salary = (SELECT MAX(salary) FROM Contracts
                  WHERE Contracts.left IS NULL);
  IF :NEW.salary >= v_max_sal THEN
    DBMS_OUTPUT.PUT_LINE('Player ' || v_ufa_id || ' requires a salary in excess of ' || :NEW.salary || ' to ensure his contractual advantage!');
  END IF;
END contracts_biu_trigger;
```

\textbf{Figure 13.1} PL/SQL mutating trigger.

In the vast majority of cases, a \textit{mutating table problem} is the result of a row-level trigger attempting to read or update the table that is the subject of the triggering event. In Figure 13.1, an attempt is made to read the contents of the \textit{contracts} table from within a row-level trigger. This trigger is itself activated via inserts or updates to that table. Such an operation is forcibly aborted by the DBMS in an attempt to maintain data integrity.

The reason for such an apparently over-restrictive exclusion is that DML operations often affect more than a single row of a table, and, as such, require an exclusive